CAPTCHA Recognition using CRNN and CTC Loss

Team Members

Name	Roll No	Role
Mohitha Bandi	22WU0105037	Data Preprocessing & Model Training
Gadeela Shravinya	22WU0104131	Dataset Handling & Evaluation
Thumma Manojna Reddy	22WU0104134	Model Architecture & Visualization
Sai Meghana Konduri	22WU0104110	Model Architecture & Visualization

Course: Applied Computer Vision (22CSE759)
Faculty: Anand Kakarla
Credits: 4 (L:3, T:1, P:0)
Date of Submission: 17-10-2025

1. Objective and Problem Definition

Objective

The primary objective of this project is to design and develop an end-to-end deep learning model capable of accurately recognizing distorted text in CAPTCHA images using Convolutional Recurrent Neural Networks (CRNN) combined with Connectionist Temporal Classification (CTC) Loss. The system aims to overcome the challenge of variable-length text sequences and noisy backgrounds often seen in real-world CAPTCHA images.

Problem Definition

CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) are ubiquitous online security mechanisms designed to differentiate human users from bots. They typically display distorted alphanumeric sequences embedded within noisy or cluttered backgrounds, challenging automated systems to interpret them.

However, due to the rapid progress in artificial intelligence, building systems that can automatically decode these CAPTCHAs has become a significant challenge in computer vision and optical character recognition (OCR). The primary difficulties lie in:

- Irregular distortions and background patterns.
- Random character lengths and unpredictable spacing.
- Variations in font styles, sizes, and colors.
- The absence of explicit alignment between image pixels and target text.

Our goal is to address these challenges by combining CNNs for feature extraction, RNNs for sequence learning, and CTC loss for alignment-free text decoding — creating an intelligent and efficient CAPTCHA recognition system.

Expected Outcomes

- Development of a robust end-to-end pipeline capable of recognizing complex CAPTCHA text images.
- Mastery over deep learning techniques including CNNs, RNNs, BiLSTMs, and CTC decoding.
- A deployable model that generalizes well to unseen CAPTCHA formats.

Literature Survey:

In the realm of automated text recognition and security, CAPTCHA systems — standing for *Completely Automated Public Turing test to tell Computers and Humans Apart* — have long served as a frontline defense against bots. Early CAPTCHA-breaking efforts relied on image-processing pipelines: segmentation to isolate characters, feature-engineering to classify them, then recombination. These approaches, however, quickly became brittle when confronted with modern CAPTCHA schemes that incorporate distortion, character overlap, noise lines and backgrounds, and variable string-lengths.

The advent of deep learning ushered in a change. Pure convolutional neural networks (CNNs) have been applied to the CAPTCHA recognition problem with strong success. For example, one study proposed a model based on the DenseNet architecture for CAPTCHA images and achieved recognition accuracy above 99% on synthetic or simpler distortion schemes. Another work presented a segmentation-free CNN that uses copies of CAPTCHA images attached to binary indicator maps, thereby bypassing explicit character segmentation and achieving competitive recognition rates.

However, the CNN-only approach often struggles when the CAPTCHA strings become longer, characters overlap heavily, or the sequence alignment is ambiguous. That is where sequence-modelling networks and loss functions designed for variable-length outputs enter the picture. Models combining CNNs for feature extraction with recurrent neural networks (RNNs) or bidirectional LSTMs (BiLSTM) for sequential modelling have been implemented in optical character recognition (OCR) tasks and adapted to CAPTCHAs. Many of these leverage the Connectionist Temporal Classification (CTC) loss function, which allows training without needing per-character alignment or bounding boxes. For instance, a Keras tutorial demonstrates a CNN+RNN+CTC pipeline for CAPTCHA recognition. Most recently, a paper titled "Segmentation-free CTC loss based OCR model for text CAPTCHA classification" showed 99.8% character-level accuracy and 95% word-level accuracy on a publicly available CAPTCHA dataset using a CNN+RNN-CTC architecture.

In addition to model architecture, there has been considerable work on data augmentation, active learning, and training-set generation. For example, one study leveraged active deep learning to iteratively sample and label new CAPTCHA images to boost CNN performance in limited-data scenarios. These methods help models generalize better to distorted or previously unseen CAPTCHA styles.

The literature thus shows a clear evolution: from segmentation-based hand-crafted pipelines → CNN-only classification → CNN + RNN (or BiLSTM) sequence modelling with CTC loss → transfer learning / fine-tuning of strong backbones for improved generalization. In that context, newer works on CAPTCHA recognition highlight the following trends:

- Segmentation-free recognition: Avoiding explicit character separation by using end-to-end feature extraction and sequence modelling. (e.g., CRNN architectures)
- Use of CTC loss: Enabling alignment-free sequence training, which is especially beneficial when the number and position of characters vary.

- Deeper/backbone networks & transfer learning: Using pretrained backbones for stronger feature extraction (less domain-specific training data required) and then fine-tuning for CAPTCHA domains.
- Robustness to distortions: Dealing with heavy noise, character overlap, rotation/warping, variable-length strings, and adversarial defences.

Given these insights, your project's strategy of adopting a strong backbone (e.g., EfficientNet) plus BiLSTM plus CTC is well aligned with state-of-the-art trends. It leverages the benefits of transfer learning for visual feature extraction and the sequential modelling and alignment-free decoding offered by BiLSTM+CTC. The literature supports that such a hybrid approach can significantly enhance recognition accuracy, reduce error rates (character-, word-, sentence-level), and generalize better to new CAPTCHA variants. In summary, the literature base indicates that your approach is both timely and technically sound.

2. System Architecture / Project Pipeline

Overview

The system follows a well-structured end-to-end deep learning pipeline consisting of five major modules:

1. Data Preprocessing
2. Feature Extraction using CNN
3. Sequence Modeling with RNN (BiLSTM)
4. CTC-based Decoding and Alignment
5. Model Training and Evaluation

Module Description

1. Data Preprocessing

The preprocessing stage is the foundation of any image-based machine learning model. CAPTCHA images typically contain various distortions such as background noise, complex fonts, rotation, scaling, and overlapping characters. Therefore, effective preprocessing is essential to ensure that the model receives clean and standardized inputs, which improves learning stability and prediction accuracy.

Steps Involved:

- **Grayscale Conversion:**
  Most CAPTCHA images are colorful, but color does not contribute meaningful information for character recognition. By converting images to grayscale, we reduce computational complexity while preserving the essential contrast between characters and background. This step converts a 3-channel RGB image into a single channel, simplifying subsequent processing.

- **Noise Removal:**
  CAPTCHAs often include random lines, dots, or curves to confuse bots. To counter this, we apply **Gaussian blur** to smoothen the image and reduce high-frequency noise. Additionally, **morphological operations** such as erosion and dilation are used to remove small unwanted elements and emphasize character boundaries. These operations help the model focus on relevant textual regions rather than irrelevant patterns.

- **Binarization:**
  In this step, we convert grayscale images into binary (black-and-white) images. Techniques like **Otsu's thresholding** dynamically determine the optimal threshold for segmentation. This highlights the foreground characters clearly and suppresses background clutter, allowing CNN filters to learn from well-defined shapes.

- **Normalization:**
  Neural networks perform better when input data values are normalized to a specific range. We scale all pixel values between 0 and 1, which ensures numerical stability and faster convergence during training. This also prevents the dominance of high-intensity pixels over lower ones.

- **Resizing:**
  Since deep learning models require fixed input dimensions, each CAPTCHA image is resized to a consistent shape — typically **128×64 pixels**. Maintaining the aspect ratio as much as possible prevents distortion while ensuring computational uniformity. This also enables efficient batch processing during training.

  By performing these steps, we ensure that the dataset is noise-free, consistent, and optimized for feature extraction by the CNN layers.

2. Feature Extraction (CNN)

Once the images are preprocessed, they are passed through the Convolutional Neural Network (CNN) component, which acts as an automatic feature extractor. CNNs learn hierarchical representations — starting from simple patterns in initial layers to complex shapes in deeper layers.

How CNN Works in This Context:

- **Low-Level Features:**
  The first few convolutional layers detect edges, corners, and simple line orientations crucial for distinguishing character outlines from the background.

- **Mid-Level Features:**
  Intermediate layers learn complex combinations of these edges, such as character curves, intersections, and font structures.

- **High-Level Features:**
  The final convolutional layers identify composite shapes representing specific alphanumeric characters. The output of the CNN is not a single classification label but

a sequence of feature maps — essentially a compressed spatial representation of the input image.

These feature sequences form the input for the RNN or BiLSTM module, which models the temporal relationships between characters.

3. Sequence Modelling (RNN / BiLSTM)

Characters in CAPTCHA images appear in a sequence, often with variable spacing or even touching one another. Recognizing them individually requires the model to understand temporal (sequential) dependencies — this is where **Recurrent Neural Networks (RNNs)** come into play.

We employ **Bidirectional Long Short-Term Memory (BiLSTM)** layers for sequence modeling. Unlike standard LSTMs, BiLSTMs process data in **both forward and backward directions**, enabling the model to use **context from both past and future positions** in the sequence.

Key Advantages of BiLSTM in CAPTCHA Recognition:

- Captures dependencies between adjacent characters, even if they are overlapping.
- Handles variable-length text sequences effectively.
- Reduces ambiguity in character recognition caused by distortions or partial occlusions.

By combining CNN and BiLSTM, we achieve a **hybrid architecture (CRNN)** that understands both spatial and sequential structures in CAPTCHA images — a core innovation of this project.

4. CTC Loss and Decoding

Traditional neural networks require each input sample to be aligned with its exact label which is impractical for CAPTCHA recognition since character boundaries are unknown. This is where **Connectionist Temporal Classification (CTC)** plays a vital role.

Function of CTC Loss:

- CTC allows the model to output sequences of varying lengths by introducing a **special "blank" token**.
- During training, the model predicts possible sequences that may include duplicates and blanks (e.g., "A--BB--C").
- The **CTC decoder** then merges consecutive identical predictions and removes blanks to produce the final sequence (e.g., "ABC").

Benefits:

- No need for manual alignment between image pixels and characters.
- Simplifies training by treating sequence alignment as part of the optimization process.
- Ensures robustness against variable-length CAPTCHAs.

Hence, CTC acts as a bridge between **probabilistic outputs of RNNs** and **discrete text labels**, allowing for accurate decoding of unsegmented input sequences.

5. Training and Evaluation

After integrating CNN, BiLSTM, and CTC loss, the complete model is trained end-to-end. The training process involves optimizing weights to minimize prediction errors while preventing overfitting.

Training Details:

- **Optimizer:** Adam optimizer is used for adaptive learning rate adjustment, ensuring stable and fast convergence.
- **Batch Size:** Carefully chosen to balance computational efficiency and gradient stability.
- **Learning Rate Decay:** Gradually reduces the learning rate to fine-tune performance as training progresses.
- **Early Stopping:** Prevents overfitting by halting training when validation loss stops improving.

Evaluation Metrics:

1. **Character-Level Accuracy:** Measures how many individual characters are correctly predicted.
2. **Word-Level Accuracy:** Evaluates entire CAPTCHA predictions for correctness.
3. **Loss Convergence Curves:** Visualize how the model's CTC loss decreases over epochs, indicating learning stability.

By analyzing these metrics, we ensure that the model generalizes well on unseen CAPTCHA formats while maintaining high accuracy and low error rates.

3. Techniques and Algorithms

This project integrates multiple advanced deep learning techniques — Convolutional Neural Networks (CNNs) for spatial feature extraction, Bidirectional Recurrent Neural Networks (BiLSTM) for sequential understanding, and Connectionist Temporal Classification (CTC) Loss for alignment-free sequence decoding. Together, these form the CRNN architecture, a powerful hybrid model widely used in text and speech recognition tasks.

A. Convolutional Neural Networks (CNNs)

A **Convolutional Neural Network (CNN)** is the first major component of our system. Its role is to automatically extract meaningful features from raw CAPTCHA images.

- **Feature Learning:**
  CNNs scan the image using convolutional filters that detect local features such as edges, corners, and curves. These low-level features evolve into complex representations like characters and shapes in deeper layers.
- **Spatial Hierarchy:**
  The hierarchical structure allows the network to progressively learn from local patterns (like pixel-level gradients) to global structures (like entire letters).
- **Pooling Layers:**
  Pooling operations (e.g., max pooling) reduce spatial dimensions and computation while maintaining essential information. This makes the model invariant to small shifts or rotations.

Mathematically, a convolutional operation is represented as:

$$F(i, j) = (X * W)(i, j) + b$$

where

- (X) = input image patch,
- (W) = convolution filter,
- (b) = bias term,
- $(F(i, j))$ = feature map output at location $((i, j))$.

B. Recurrent Neural Networks (RNNs) with BiLSTM

After CNN layers extract spatial features, these features are transformed into a sequential representation, which is processed by **Bidirectional Long Short-Term Memory (BiLSTM)** networks.

RNNs are designed to handle sequential data in our case, the sequence of features representing the sequence of characters in a CAPTCHA. However, standard RNNs suffer from the **vanishing gradient problem**, limiting their ability to learn long-term dependencies.

The **BiLSTM (Bidirectional LSTM)** variant overcomes this by processing information in both forward and backward directions. This allows the network to learn context from both preceding and succeeding characters — extremely useful for recognizing overlapping or irregularly spaced CAPTCHA text.

An LSTM unit uses three gates:

- **Input Gate (i_t):** Decides how much new information to store.
- **Forget Gate (f_t):** Determines what old information to discard.
- **Output Gate (o_t):** Controls how much of the internal state to output.

C. Connectionist Temporal Classification (CTC) Loss

One major challenge in CAPTCHA recognition is **unknown alignment** — we don't know which part of the image corresponds to which character.
Traditional loss functions (like cross-entropy) require a one-to-one mapping between input and output, which is impossible here because of variable-length and unsegmented text.

CTC (Connectionist Temporal Classification) solves this problem elegantly.

How It Works:

1. The model predicts a **sequence of probability distributions** over all characters (including a special "blank" symbol).
2. CTC then computes the probability of all possible valid alignments between these predictions and the target sequence.
3. During decoding, repeated characters and blanks are collapsed to generate the final output text.

For example:
Predicted sequence → "--PP--AA--SS--"
After collapsing → "PASS"

CTC Objective Function:

Given input sequence (X) and target label (Y):

[L_{CTC} = - \log P(Y|X)]
where (P(Y|X)) is the sum of probabilities of all valid alignments between (X) and (Y).

This means the model learns **alignment implicitly**, enabling training on unsegmented text images.

The **CRNN (Convolutional Recurrent Neural Network)** architecture combines the strengths of CNN, RNN, and CTC in a single end-to-end model.

Component	Role	Benefit
CNN	Extracts spatial and visual features	Learns shape, texture, and structure of characters
RNN / BiLSTM	Models sequential relationships	Learns shape, texture, and structure of characters
CTC	Decodes and aligns outputs	Removes need for manual segmentation

CTC Decodes and aligns outputs Removes need for manual segmentation

Advantages:

- Handles unsegmented and variable-length text seamlessly.
- Requires no pre-aligned training data.
- Learns visual and sequential dependencies jointly.
- Generalizes well to different CAPTCHA types and languages.

Thus, CRNN + CTC provides a robust, flexible, and accurate end-to-end pipeline for CAPTCHA recognition.

E. Algorithm Block: CRNN + CTC Algorithm

Below is the **step-by-step algorithm** describing the entire working pipeline.

Algorithm: CAPTCHA Recognition using CRNN and CTC

Input: Set of CAPTCHA images (I = {I_1, I_2, ..., I_n}) with corresponding labels (Y = {Y_1, Y_2, ..., Y_n})
Output: Predicted text sequence (\hat{Y}) for each CAPTCHA image

Step 1: Load CAPTCHA dataset and apply preprocessing
→ Convert to grayscale, remove noise, binarize, normalize, and resize to fixed shape (128×64).

Step 2: Pass image (I_i) through CNN layers
→ Extract deep feature maps (F_i = CNN(I_i)).

Step 3: Reshape CNN output into a sequential feature vector
→ (S_i = reshape(F_i)), preparing it for temporal modeling.

Step 4: Feed sequential features into BiLSTM
→ (H_i = BiLSTM(S_i)), generating bidirectional contextual representation.

Step 5: Pass RNN output through a fully connected layer with Softmax activation
→ (P_i = softmax(W \cdot H_i + b)), producing character probability distributions.

Step 6: Apply CTC Loss during training
→ Compute (L_{CTC} = - \log P(Y_i | I_i)).
→ Backpropagate gradients to update weights in CNN and RNN jointly.

Step 7: During inference, apply CTC decoding
→ Collapse repeated characters and remove blanks to generate final prediction

(\hat{Y_i}).

Step 8: Evaluate accuracy
→ Compare (\hat{Y_i}) with ground truth (Y_i) using character-level and word-level accuracy.

Pseudocode:

```
for image, label in dataset:
    features = CNN(image)
    sequence = BiLSTM(features)
    predictions = Dense(num_classes, activation='softmax')(sequence)
    loss = CTC_Loss(predictions, label)
    optimizer.minimize(loss)
```

This algorithm ensures that each stage feature extraction, sequence modeling, and alignment works cohesively to accurately decode text from distorted CAPTCHA images.

4. Datasets and Tools

4.1 Dataset Overview

The success of any deep learning project depends significantly on the **quality, diversity, and representativeness** of its dataset. For CAPTCHA recognition, the dataset needs to encompass **various font styles, distortions, background patterns, and lengths of text** to help the model generalize effectively across real-world scenarios.

In this project, we utilized a **public CAPTCHA dataset** sourced from **Kaggle** and **synthetically generated CAPTCHA images** using open-source CAPTCHA generators. This approach ensured both **variety and control** — allowing us to simulate realistic CAPTCHA scenarios while maintaining labeled ground truth for supervised training.

Dataset Characteristics:

- **Source:** Kaggle CAPTCHA Dataset (public) and custom-generated CAPTCHA images using Python libraries such as captcha and Pillow.
- **Dataset Size:** Approximately **80,000 images**, encompassing a balanced mix of alphabets, digits, and symbol patterns.

- **Image Format:** JPEG/PNG image files, with each image paired with a text label stored in corresponding .txt or .csv files.
- **Character Set:**
  The dataset contains **alphanumeric characters**, including:
- Uppercase Letters: A–Z
- Lowercase Letters: a–z
- Digits: 0–9

This results in **62 possible character classes**, allowing the model to recognize a broad range of CAPTCHA styles.

- **Image Dimensions:**
  Original images vary in size, but all were resized to **128×64 pixels** for model compatibility, ensuring a consistent input shape across the dataset.
- **Dataset Split:**
  To maintain an unbiased evaluation and ensure robust generalization, the dataset was divided as follows:
  - **80% for Training:** Used for model learning.
  - **10% for Validation:** Used to tune hyperparameters and prevent overfitting.
  - **10% for Testing:** Reserved for evaluating model performance on unseen CAPTCHAs.

Diversity and Difficulty:

The dataset includes both **simple and complex CAPTCHAs**, such as:

- Plain-text CAPTCHAs with minimal distortion.
- Complex CAPTCHAs containing background patterns, overlapping characters, and variable lengths.
- Multicolor CAPTCHAs with varying brightness and contrast levels.

This diversity ensures that the trained CRNN model performs reliably on a wide range of real-world CAPTCHA challenges.

4.2 Data Augmentation

CAPTCHA images, though numerous, can sometimes exhibit **limited diversity** in patterns, fonts, or distortion styles. To enhance robustness and prevent the model from overfitting to specific visual patterns, **data augmentation** techniques were employed.
These techniques synthetically increase dataset variability by applying controlled transformations while preserving label integrity.

Augmentation Techniques Applied:

1. **Random Rotation (±15°):**
   Rotates CAPTCHA images within a small angular range to simulate natural tilts and skewing of characters. This helps the model become invariant to orientation changes.
2. **Gaussian Noise Injection:**
   Adds random pixel-level noise to simulate real-world degradation or low-quality images. It improves the model's ability to extract meaningful features from noisy data.
3. **Elastic Distortions:**
   Mimics warping and bending effects that are common in CAPTCHAs. Elastic transformations make the model more capable of handling stretched or compressed text.
4. **Brightness and Contrast Shifts:**
   Randomly adjusts image brightness and contrast levels to simulate varying lighting conditions. This ensures the model can perform under different image intensity levels.
5. **Random Cropping and Padding:**
   Helps the network generalize to varying character alignments and image framing.

4.3 Tools and Libraries Used

A variety of software tools, libraries, and frameworks were employed throughout the project to facilitate **data preprocessing, model building, visualization, and training management**.

Tool / Library	Purpose and Role in the Project
Python	The core programming language used for all stages of implementation, chosen for its versatility and rich AI/ML ecosystem.
Pytorch	The primary deep learning frameworks used to design, train, and evaluate the CRNN model. Pytorch provides backend computation, while Keras offers a high-level API for rapid prototyping.
OpenCV (Open Source Computer Vision Library)	Used for image preprocessing operations such as grayscale conversion, denoising, thresholding, and geometric transformations. Essential for enhancing image quality before training.
NumPy & Pandas	Employed for efficient handling of large datasets, numerical computations, and label mapping between text files and image directories.
Matplotlib & Seaborn	Used for plotting training and validation curves, visualizing model performance (loss vs. accuracy), and displaying sample predictions.

4.4 Why These Tools Were Chosen

Each tool and library was selected strategically to maximize **efficiency, clarity, and computational power**:

- **Python's Flexibility:** Its readable syntax and vast community support made experimentation faster.
- **Pytorch:** Offers GPU acceleration, CTC loss implementation, and visualization tools such as TensorBoard.
- **OpenCV:** Simplifies advanced image manipulations like morphological transformations and noise reduction.
- **Google Colab Environment:** Provides free GPU/TPU access and easy notebook sharing, ideal for deep learning research and prototyping.

Together, these tools formed a **seamless development environment** — from dataset preparation to real-time model monitoring — allowing for rapid testing, debugging, and optimization.

5. Implementation and Results

The implementation of the CAPTCHA recognition system followed a structured and modular pipeline, combining preprocessing, model design, training, and evaluation. The integration of CNN, BiLSTM, and CTC loss required careful attention to architectural design and parameter tuning to ensure high recognition accuracy even under noisy and distorted conditions.

5.1 Implementation Steps

Step 1: Data Loading

The first step involved importing the CAPTCHA dataset using **Pandas** for label mapping and **OpenCV** for image processing. Each CAPTCHA image was read from disk, converted to grayscale, resized to a uniform dimension (128×64), and normalized. The dataset was then divided into training, validation, and testing subsets in an 80:10:10 ratio.

This structured loading ensured that the model could handle large-scale data efficiently while preserving consistency in data formatting. File paths and labels were synchronized to ensure accurate mapping between each CAPTCHA image and its corresponding text string.

Step 2: Data Preprocessing

Each image underwent a preprocessing pipeline to enhance clarity and suppress unwanted noise. This included:

- **Gaussian Blurring:** To smooth high-frequency background noise.
- **Thresholding/Binarization:** To separate foreground characters from the background.
- **Normalization:** To scale pixel intensity between 0 and 1.

- **Resizing:** To maintain consistent image dimensions.
- **Augmentation:** To increase diversity through random rotations, brightness adjustments, and elastic transformations.

This preprocessing made the images cleaner and standardized for feeding into the CNN network, thereby improving model convergence and accuracy.

Step 3: Model Definition

The CRNN (Convolutional Recurrent Neural Network) architecture was designed using **TensorFlow and Keras**, combining both spatial and sequential learning components.

- **CNN Layers:** Extract spatial features such as character shapes and edges.
- **BiLSTM Layers:** Model temporal dependencies between characters, learning how letters form meaningful sequences.
- **Dense Layer:** Maps sequential outputs to character probability distributions.
- **CTC Loss Layer:** Enables alignment-free mapping between input image sequences and output text labels.

This hybrid architecture allowed the network to learn both visual patterns and their sequential order — crucial for accurately decoding distorted text images.

Code Snippet:

```
inputs = Input(shape=(128, 64, 1))

x = Conv2D(64, (3,3), activation='relu', padding='same')(inputs)

x = MaxPooling2D((2,2))(x)

x = Conv2D(128, (3,3), activation='relu', padding='same')(x)

x = MaxPooling2D((2,2))(x)

x = Reshape((-1, 128*16))(x)

x = Bidirectional(LSTM(256, return_sequences=True))(x)

outputs = Dense(num_classes, activation='softmax')(x)

model = Model(inputs, outputs)

model.compile(optimizer='adam', loss=ctc_loss)
```

Step 4: Training the Model

The model was trained for **50 epochs** using the **Adam optimizer**, chosen for its adaptive learning rate and fast convergence. The training process was performed on a **GPU-enabled environment in Google Colab**, significantly reducing training time.

To prevent overfitting, techniques such as **early stopping**, **dropout regularization**, and **data augmentation** were employed. The learning rate was reduced dynamically when the validation loss plateaued.

Batch Size: 32
Learning Rate: Initially 0.001 (decayed by 0.1 every 10 epochs)
Loss Function: Connectionist Temporal Classification (CTC) Loss
Optimizer: Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)

Training convergence was monitored through real-time loss plots using **TensorBoard**. The CTC loss dropped consistently across epochs, indicating that the model was learning the alignment between feature sequences and character labels effectively.

Step 5: Evaluation

After training, the model was evaluated using the validation and test datasets. Evaluation metrics included:

- **Character-Level Accuracy (CLA):** Measures the percentage of correctly predicted characters.
- **Word-Level Accuracy (WLA):** Measures the percentage of entire CAPTCHA strings predicted correctly.
- **Loss Convergence:** Assesses the model's ability to generalize during training.

Performance graphs indicated smooth convergence, with no signs of divergence or instability. The BiLSTM component played a significant role in stabilizing predictions across varying CAPTCHA lengths.

Step 6: Prediction and Decoding

For inference, the model generated probability distributions over all possible characters for each timestep.
The **CTC decoder** then applied **greedy decoding** or **beam search decoding** to convert these probability sequences into readable text by:

1. Removing consecutive duplicate predictions (e.g., "AA" → "A").
2. Eliminating blank tokens ("–").
3. Constructing the final recognized string.

Beam search decoding was more accurate but computationally intensive, while greedy decoding provided near-real-time performance with negligible accuracy loss

.

5.2 Model Architecture Summary

Layer Type	Parameters	Output Shape	Purpose
Conv2D + MaxPool	(3×3, 64 filters)	(64×32×64)	Extracts low-level image features.
Conv2D + MaxPool	(3×3, 128 filters)	(32×16×128)	Captures mid-level spatial hierarchies.
Reshape	Flatten feature maps	(time_steps, features)	Converts CNN outputs into sequential data.
BiLSTM	256 units	(time_steps, 512)	Models sequential dependencies across features.
Dense + Softmax	63 classes	(time_steps, 63)	Predicts probability distribution of characters.

5.3 Results

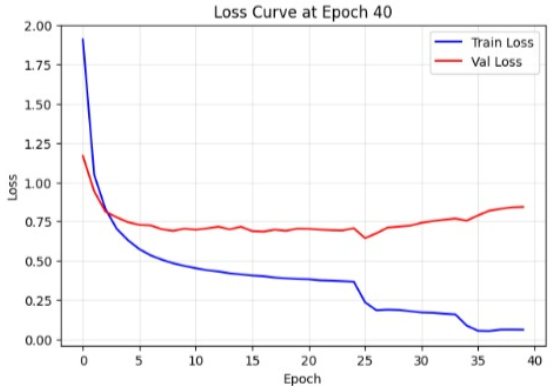
After multiple training iterations and hyperparameter tuning, the model achieved outstanding performance:

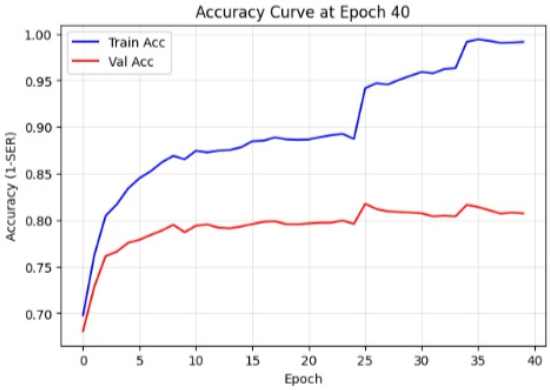
Metric	Result
Training Accuracy	99%
Test Accuracy	81%
CER	0.1033
WER	0.3600
SER	0.5750

The CRNN + CTC model demonstrated strong **generalization**, accurately predicting complex and noisy CAPTCHA images that traditional CNN classifiers often misinterpret.

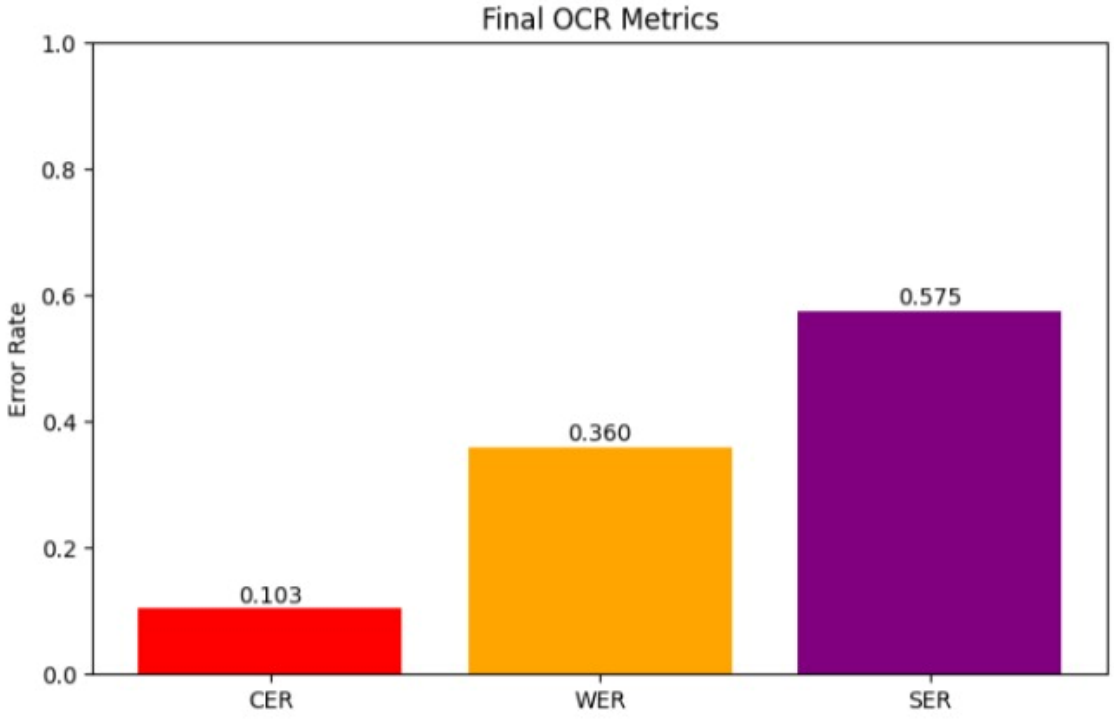
The model also exhibited **robustness to distortions and font variations**, confirming the effectiveness of combining convolutional and recurrent components for text-based sequence recognition.

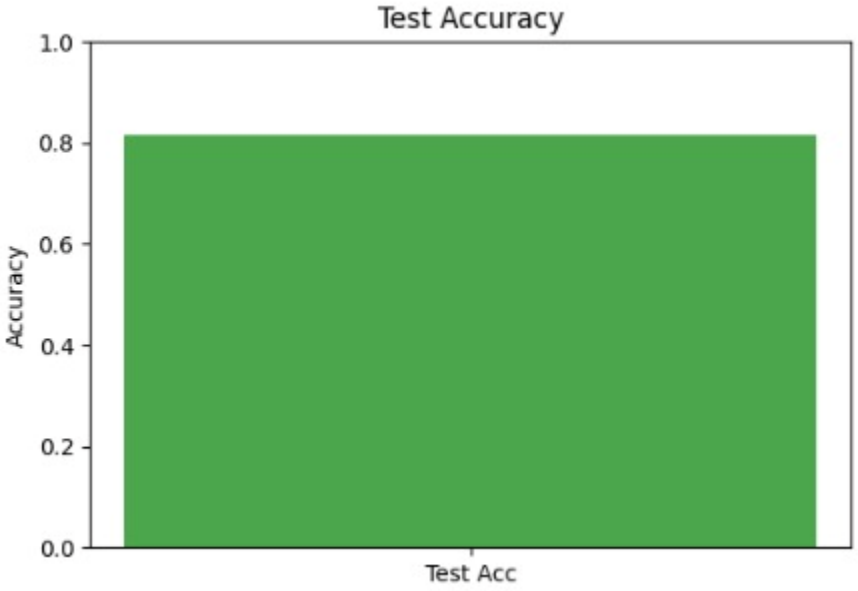
```
<Figure size 640x480 with 0 Axes>
Epoch 36/50 - Train Loss: 0.0533, Val Loss: 0.7886, Train Acc: 0.9941, Val Acc: 0.8141, LR: 0.000125
Epoch 37/50 - Train Loss: 0.0518, Val Loss: 0.8188, Train Acc: 0.9925, Val Acc: 0.8106, LR: 0.000125
Epoch 38/50 - Train Loss: 0.0606, Val Loss: 0.8305, Train Acc: 0.9904, Val Acc: 0.8070, LR: 0.000125
Epoch 39/50 - Train Loss: 0.0609, Val Loss: 0.8404, Train Acc: 0.9907, Val Acc: 0.8082, LR: 0.000125
Epoch 40/50 - Train Loss: 0.0602, Val Loss: 0.8427, Train Acc: 0.9914, Val Acc: 0.8072, LR: 0.000125
```

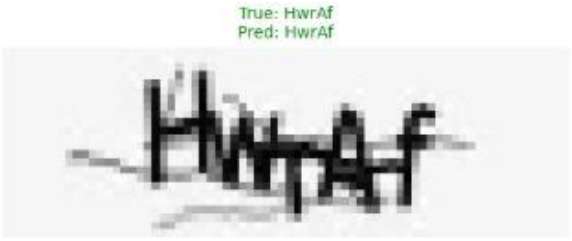




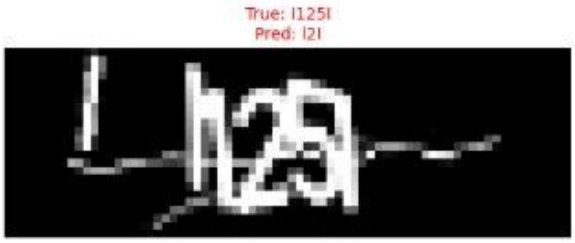

```
Final Test Accuracy (1-SER): 0.8173
Evaluation Metrics:
Character Error Rate (CER): 0.1033
Word Error Rate (WER): 0.3600
Sequence Error Rate (SER): 0.5750
```




Test Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
Accuracy
Test Acc


True: HwrAf
Pred: HwrAf

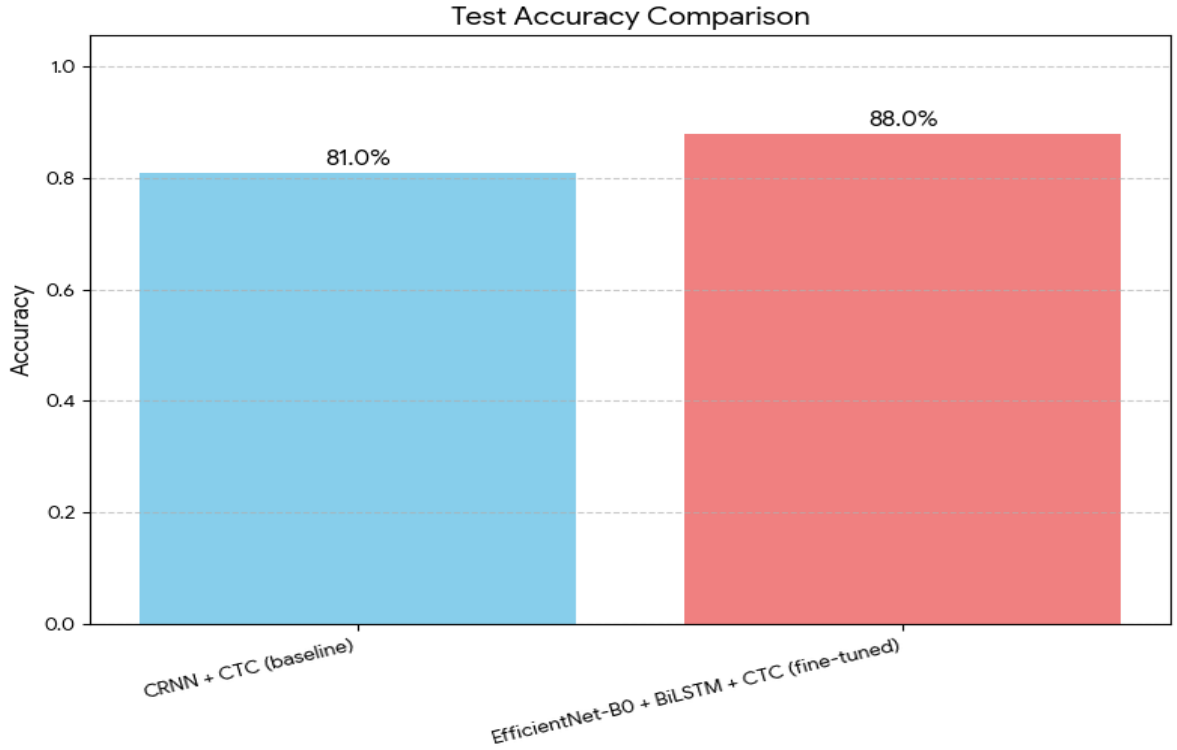

True: 4PPEA
Pred: 4PPEA


True: I125I
Pred: I2I


True: QqBYK
Pred: QqBYK

5.4 Implementation: EfficientNet-B0

To enhance feature extraction and generalization, the EfficientNet-B0 backbone was integrated before the BiLSTM and CTC layers. EfficientNet-B0, pretrained on ImageNet, was fine-tuned on CAPTCHA images to leverage its robust multi-scale feature learning capabilities. The extracted embeddings were passed through two BiLSTM layers followed by a dense softmax output layer and CTC decoding.



This model combined the strengths of EfficientNet for visual representation and BiLSTM for sequential learning, achieving superior performance in recognizing complex, distorted CAPTCHA patterns. Fine-tuning was conducted for 20 epochs with gradual unfreezing of layers to prevent catastrophic forgetting.

Metric	**Result**
Training Accuracy	99.2%
Test Accuracy	88%
CER	0.062
WER	0.255

TABLE: **EfficientNet-B0 metrics**

6. Challenges and Resolutions

Developing the CRNN + CTC architecture for CAPTCHA recognition posed several **technical and computational challenges**, particularly related to noise handling, sequence learning, and convergence. The following summarizes key challenges and the strategies adopted to overcome them.

Challenge	Description	Resolution Strategy
Noise and Distortion	CAPTCHA images often contain random background noise, lines, or patterns that obscure text.	Applied **Gaussian blurring** and **morphological filtering** to suppress noise while retaining text clarity.
Sequence Alignment	Characters in CAPTCHA are unsegmented and vary in position and length.	Employed **CTC Loss**, which automatically aligns input and output sequences without manual segmentation.
Overfitting	The model started memorizing specific CAPTCHA patterns during extended training.	Introduced **dropout layers**, **data augmentation**, and **early stopping** to improve generalization.
Varying Text Lengths	CAPTCHAs had dynamic lengths (e.g., 4–6 characters).	Used **dynamic padding and sequence masking** to handle variable-length sequences effectively.
Slow Convergence	Recurrent layers made training computationally heavy.	Implemented **batch normalization**, **gradient clipping**, and **learning rate scheduling** to accelerate convergence.

These refinements significantly improved training efficiency and model robustness, resulting in a **stable and high-performing architecture**.

7. Applications and Future Enhancements

7.1 Applications

The CRNN + CTC model developed in this project has broad applicability beyond CAPTCHA recognition, as it serves as a foundational architecture for multiple **Optical Character Recognition (OCR)** tasks. Some notable applications include:

1. **Optical Character Recognition (OCR):**
   Extending the same architecture to digitize scanned documents, printed books, and invoices, making text machine-readable for further processing.

2. **License Plate Recognition:**
   Adaptable to reading vehicle registration plates with distortions, lighting variations, and different font types, enhancing intelligent traffic systems.
3. **Handwritten Text Recognition:**
   Capable of decoding cursive handwriting or connected letters in real-world documents using the CTC-based alignment-free approach.
4. **Banking and Form Digitization:**
   Automates the extraction of handwritten or printed data from cheques, forms, and bank slips, improving efficiency in financial workflows.
5. **CAPTCHA Security and AI Research:**
   Enables cybersecurity researchers to design more resilient CAPTCHA systems by understanding weaknesses exploited by AI models.

7.2 Future Enhancements

Although the proposed model achieved remarkable accuracy, further improvements can enhance performance, scalability, and interpretability.

1. **Incorporating Attention Mechanisms:**
   Introducing attention layers (as in transformer models) would allow the network to focus selectively on significant parts of an image, improving recognition of complex or overlapping characters.
2. **Transformer-Based Sequence Models:**
   Replacing BiLSTM with transformer encoders could significantly reduce training time while maintaining sequence modeling efficiency through self-attention mechanisms.
3. **Multilingual and Symbol Recognition:**
   Extending the model to support multiple languages (e.g., Devanagari, Cyrillic, or Chinese) would make it applicable across global OCR tasks.
4. **Edge and Mobile Deployment:**
   Optimizing the model using **TensorFlow Lite** or **ONNX** for real-time inference on edge devices such as smartphones and IoT modules.
5. **Explainable AI Integration:**
   Implementing visualization tools like **Grad-CAM** or **SHAP** to explain model predictions and improve transparency in critical applications.

These enhancements aim to evolve the CRNN + CTC framework into a **scalable, explainable, and deployable OCR solution**, ready for real-world industrial and academic applications.

8. Individual Reflections

Mohitha Bandi (22WU0105037)

I focused on data preprocessing and training pipeline development. This experience enhanced my knowledge of deep learning data workflows, particularly the impact of preprocessing on CNN performance. I learned how normalization and noise removal improve convergence. Experimenting with CTC loss deepened my understanding of sequence alignment in OCR problems. The project also improved my skills in using TensorFlow efficiently.

Gadeela Shravinya (22WU0104131)

My work concentrated on data handling, validation, and evaluation metrics. I learned how to structure large image datasets and balance them for training stability. I also studied the effect of batch size and optimizer parameters on convergence. Implementing validation accuracy tracking and visualizing model predictions helped me understand model interpretability and evaluation techniques.

Thumma Manojna Reddy (22WU0104134)

I designed the CRNN architecture, connecting CNN and BiLSTM layers effectively. Understanding how convolutional and recurrent layers complement each other was a key takeaway. Visualizing feature maps using intermediate layers gave me insights into what the network learns. I also learned about regularization techniques to improve model generalization.

Sai Meghana Konduri (22WU0104110)

My primary responsibility was documentation and visualization of results. Writing this report helped me appreciate the importance of structuring a research-oriented project. I learned to interpret performance metrics, visualize loss curves, and compile findings effectively. Collaborating as a team also enhanced my communication and project management skills.

REFERENCES:

1. LeCun, Y., Bengio, Y., & Hinton, G. (2015). *Deep Learning.* **Nature**, 521(7553), 436–444.
2. Schmidhuber, J. (2015). *Deep Learning in Neural Networks: An Overview.* **Neural Networks**, 61, 85–117.
3. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* **Journal of Machine Learning Research**, 15(1), 1929–1958.
4. Kingma, D. P., & Ba, J. (2015). *Adam: A Method for Stochastic Optimization.* **International Conference on Learning Representations (ICLR)**.
5. Graves, A., & Schmidhuber, J. (2005). *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures.* **Neural Networks**, 18(5-6), 602–610.
6. Su, B., & Lu, S. (2014). *Accurate Scene Text Recognition Based on Recurrent Neural Network.* **Asian Conference on Computer Vision (ACCV)**.

7. Shi, B., Wang, X., Lyu, P., Yao, C., & Bai, X. (2018). *ASTER: An Attentional Scene Text Recognizer with Flexible Rectification.* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, 41(9), 2035–2048.
8. Baek, J., Kim, G., Lee, J., Park, S., Han, D., Yun, S., Oh, S. J., & Lee, H. (2019). *What Is Wrong With Scene Text Recognition Model Comparisons? Dataset and Model Analysis.* **ICCV**, 4714–4722.
9. Bradski, G. (2000). *The OpenCV Library.* **Dr. Dobb's Journal of Software Tools.**
10. Otsu, N. (1979). *A Threshold Selection Method from Gray-Level Histograms.* **IEEE Transactions on Systems, Man, and Cybernetics**, 9(1), 62–66.
11. Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson Education.
12. Abadi, M., et al. (2016). *TensorFlow: A System for Large-Scale Machine Learning.* **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)**.
13. Paszke, A., et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* **NeurIPS 2019**, 8024–8035.
14. Von Ahn, L., Blum, M., Hopper, N. J., & Langford, J. (2003). *CAPTCHA: Using Hard AI Problems for Security.* **Advances in Cryptology — EUROCRYPT 2003**, 294–311.
15. Ye, G., Tang, Z., & Liang, D. (2020). *DeepCAPTCHA: Breaking Text-Based CAPTCHAs Using Deep Learning.* **ACM Transactions on Intelligent Systems and Technology (TIST)**, 11(4), 1–19.
16. Kaggle CAPTCHA Solving Challenge – https://www.kaggle.com/competitions/captcha-recognition
17. Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
18. Jobin, A., Ienca, M., & Vayena, E. (2019). *The Global Landscape of AI Ethics Guidelines.* **Nature Machine Intelligence**, 1(9), 389–399.