Text Recognition with CRNN and CTC for Sequence Character Recognition

CS 517: Introduction to Human Computer Interaction
Project 2 (Fall 2024)

Kuladeep[1], Hemanth[2], and Saiprakash[3]

[1]B01058870 cgupta2@binghamton.edu
[2]B01043253 hborra@binghamton.edu
[3]B01037579 snalubolu@binghamton.edu

October 20, 2024

Abstract

This report presents the implementation of a Convolutional Recurrent Neural Network (CRNN) to recognize sequences of alphabets and digits from images. The tasks involved developing and evaluating a custom CRNN with both Bidirectional GRU and LSTM for sequential text recognition, fine-tuning pre-trained models, and recognizing handwritten sequences from a video. We provide a comprehensive analysis of the performance metrics, optimizers, learning rates, and pre-trained model fine-tuning strategies.

1 Introduction

Text recognition is a key application in computer vision, useful in automated data extraction from documents. The project focuses on recognizing sequential data in images, particularly text from scanned documents or photos, without requiring detailed segmentation of individual characters. The design of the system system includes CRNN and the Connectionist Temporal Classification (CTC) loss function. The primary tasks involve custom CRNN implementation, comparing RNN units (GRU vs. LSTM), experimenting with different optimizers and learning reates, fine-tuning pre-trained models(Resnet50), and evaluating system performance on video data.
This project explores various approaches to enhance the accuracy and efficiency of character and digit recognition systems. The methodology is divided into three primary tasks:

1. Implementation of customized Convolutional Recurrent Neural Networks (CNNs) trained on a dataset consisting of 600,000 images. Comparing the results between GRU and LSTM for RNN layers. Analysis of different optimizers(Adam, SGD) and learning rates.

2. Fine-tuning pre-trained model to replace CNN structure and then connect to LSTM/GRU for improved performance.

3. Extending the model's applicability by recognizing handwritten words and digits from video frames using OpenCV.

Dataset: We used a subset of the MJ Synth Dataset, reduced to one-tenth of its original size. The dataset contains images of English digits and words, with standard training, validation, and test splits. This diverse dataset allowed us to explore sequence recognition, making it suitable for model experimentation and evaluation. Due to resource constraints, we used 300,000 images for the experiments. The dataset was split into 240,000 images for training and 60,000 images for validation. This subset allowed us to conduct meaningful training and validation while managing GPU and memory limitations effectively.

Data pre-processing:

- **Image Normalization and Resizing**: Images were resized to 224x224 pixels and converted to RGB. Normalization was applied using standard RGB mean and standard deviation values, ensuring consistency in input, which helps in stabilizing training.
- **DataLoader setup**: Preprocessed images were used to create custom datasets for training and validation. Data loaders were used for efficient data batching, with shuffling applied to training data
- **Collate funcion**: A custom collate function was used to handle variable-length sequences by stacking images and concatenating labels, ensuring uniform batches for training with the CTC loss.

The following sections detail the methodologies, experimental setups, results, and comparative analyses of these tasks.

2 Task 1: Customized CRNN immplementation

2.1 Architecture Design

The Convolutional Recurrent Neural Network (CRNN) implemented for Task 1 comprises a series of convolutional layers followed by bidirectional recurrent layers. The architecture is designed to extract spatial features from images and then capture sequential dependencies, making it highly suitable for text sequence recognition.

- **Convolution Layers**: Extract spatial features, reduce dimensionality with pooling, and use Batch Normalization for stable learning. The layers start with 64 filters and increase up to 512 filters. A total of 7 Convolution layers were used.
- **Recurrent Layers**: The bidirectional layers are implemented using either LSTM or GRU. Each RNN outputs a hidden state of size 256, giving the network the ability to model temporal dependencies effectively.
- **Fully Connected Layers**: Projects the RNN outputs to character classes, representing the characters that the network can predict

This CRNN architecture is powerful for recognizing text sequences in images, as it effectively combines convolutional feature extraction with recurrent sequence modeling, making it capable of handling the spatial and sequential complexity of text in images. A visual representation of the CNN architecture is shown in Figure 1.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CRNN                                     [26, 1, 37]               --
├─Sequential: 1-1                        [1, 512, 1, 26]           --
│    └─Conv2d: 2-1                       [1, 64, 32, 100]          640
│    └─ReLU: 2-2                         [1, 64, 32, 100]          --
│    └─MaxPool2d: 2-3                    [1, 64, 16, 50]           --
│    └─Conv2d: 2-4                       [1, 128, 16, 50]          73,856
│    └─ReLU: 2-5                         [1, 128, 16, 50]          --
│    └─MaxPool2d: 2-6                    [1, 128, 8, 25]           --
│    └─Conv2d: 2-7                       [1, 256, 8, 25]           295,168
│    └─ReLU: 2-8                         [1, 256, 8, 25]           --
│    └─BatchNorm2d: 2-9                  [1, 256, 8, 25]           512
│    └─Conv2d: 2-10                      [1, 256, 8, 25]           590,080
│    └─ReLU: 2-11                        [1, 256, 8, 25]           --
│    └─MaxPool2d: 2-12                   [1, 256, 4, 26]           --
│    └─Conv2d: 2-13                      [1, 512, 4, 26]           1,180,160
│    └─ReLU: 2-14                        [1, 512, 4, 26]           --
│    └─BatchNorm2d: 2-15                 [1, 512, 4, 26]           1,024
│    └─Conv2d: 2-16                      [1, 512, 4, 26]           2,359,808
│    └─ReLU: 2-17                        [1, 512, 4, 26]           --
│    └─MaxPool2d: 2-18                   [1, 512, 2, 27]           --
│    └─Conv2d: 2-19                      [1, 512, 1, 26]           1,049,088
│    └─ReLU: 2-20                        [1, 512, 1, 26]           --
├─Sequential: 1-2                        [26, 1, 512]              --
│    └─BidirectionalRNN: 2-21            [26, 1, 512]              --
│    │    └─LSTM: 3-1                    [26, 1, 512]              1,576,960
│    │    └─Linear: 3-2                  [26, 512]                 262,656
│    └─BidirectionalRNN: 2-22            [26, 1, 512]              --
│    │    └─LSTM: 3-3                    [26, 1, 512]              1,576,960
│    │    └─Linear: 3-4                  [26, 512]                 262,656
├─Linear: 1-3                            [26, 1, 37]               18,981
==========================================================================================
Total params: 9,248,549
Trainable params: 9,248,549
Non-trainable params: 0
Total mult-adds (M): 729.77
==========================================================================================
```

Figure 1: Customized CRNN Architecture

Additionally, a snippet of the network definition is shown below to illustrate the structure:

```
class CRNN(nn.Module):
    def __init__(self, imgH, nc, nclass, nh, rnn_type='LSTM'):
        super(CRNN, self).__init__()
        assert imgH % 16 == 0, 'imgH has to be a multiple of 16'

        # CNN layers
        self.cnn = nn.Sequential(
            nn.Conv2d(nc, 64, 3, 1, 1),  # Conv1
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),          # Pool1

            nn.Conv2d(64, 128, 3, 1, 1), # Conv2
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),          # Pool2

            nn.Conv2d(128, 256, 3, 1, 1),# Conv3
```

```
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(256),
            nn.Conv2d(256, 256, 3, 1, 1),# Conv4
            nn.ReLU(inplace=True),
            nn.MaxPool2d((2,2), (2,1), (0,1)),  # Pool3

            nn.Conv2d(256, 512, 3, 1, 1),# Conv5
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(512),
            nn.Conv2d(512, 512, 3, 1, 1),# Conv6
            nn.ReLU(inplace=True),
            nn.MaxPool2d((2,2), (2,1), (0,1)),  # Pool4

            nn.Conv2d(512, 512, 2, 1, 0),# Conv7
            nn.ReLU(inplace=True)
        )

        self.rnn = nn.Sequential(
            BidirectionalRNN(512, nh, rnn_type=rnn_type),
            BidirectionalRNN(nh * 2, nh, rnn_type=rnn_type)
        )

        self.embedding = nn.Linear(nh * 2, nclass)

    def forward(self, x):
        # x: (batch_size, channels, height, width)
        conv = self.cnn(x)
        b, c, h, w = conv.size()
        assert h == 1, "the height of conv must be 1"
        conv = conv.squeeze(2)  # Remove height dimension, now (
           batch_size, channels, width)
        conv = conv.permute(2, 0, 1)  # (width, batch_size,
           channels)

        rnn_output = self.rnn(conv)  # (width, batch_size, nh *
           2)
        output = self.embedding(rnn_output)  # (width, batch_size
           , nclass)
        return output
```

Listing 1: Custom CNN Architecture Snippet

2.2 Hyperparameter Configuration

Key hyperparameters were fixed to ensure consistent training across different experiments:

- **Learning Rate**: 0.0001
- **Number of Layers**: 10 layers (7 convolutional, 2 Bidirectional recurrent layers, 1 fully connected layer)
- **Total number of parameters in the model**: 9,248,549

These hyperparameters were chosen based on preliminary experiments to strike a balance between training speed and model performance.

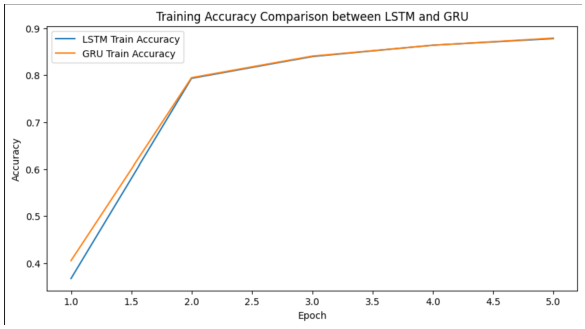
2.3 Optimizers and Learning rates

Initially to compare GRU and LSTM we used 0.0001 as common learning rate and Adam as optimizer to analyze the difference in LSTM and GRU. Since LSTM and GRU had similar accuracies after 5 epochs, we used GRU with different optimizers(Adam, SGD) and learning rates(0.001, 0.05) to analyze the impact of each on the model.

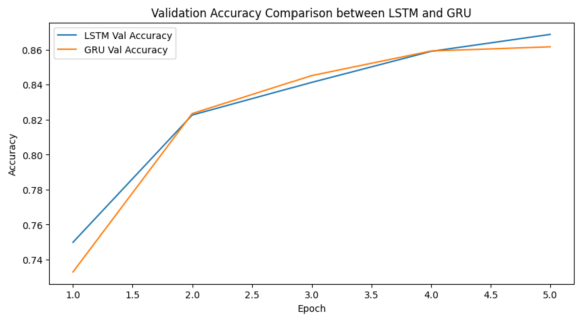
- **Optimizers**:
  1. Adam
  2. Stochastic Gradient Descent (SGD)

2.4 Experimental Results

Training and testing were conducted over five epochs for each configuration. The performance metrics, including training accuracy, Validation accuracy, and Training loss, Validation loss were recorded. Figures 2a, 2b, 2c, and 2d illustrate the comparison of test and training accuracies and losses of GRU and LSTM. Figures 3a, 3b, 3c, and 3d illustrate the comparison of test and training accuracies and losses of different optimizers and learning rates used on GRU.



(a) Training Accuracy Comparison between LSTM and GRU

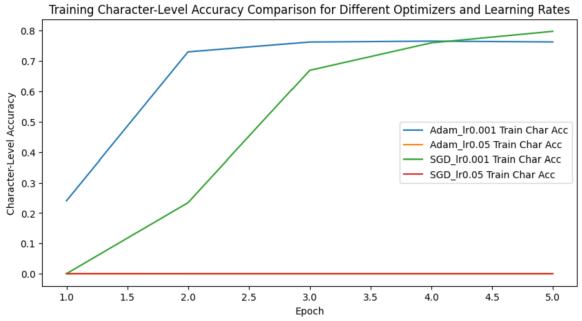


(b) Validation Accuracy Comparison between LSTM and GRU

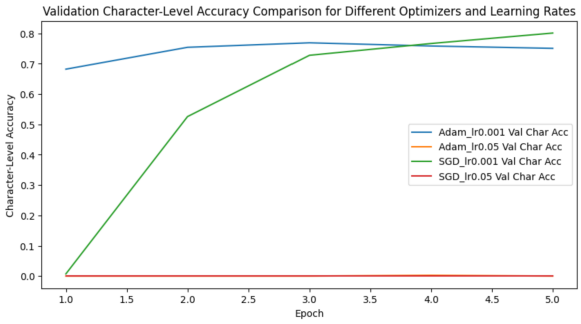
(c) Training Loss Comparison between LSTM and GRU

(d) Validation Loss Comparison between LSTM and GRU

Figure 2: Training and Validation Performance Comparison between LSTM and GRU

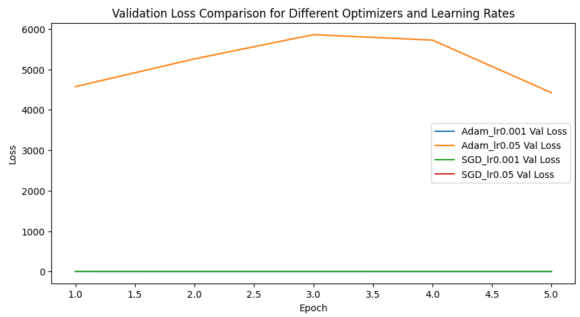


(a) Training Accuracy Comparison between different optimizers and Learning rates



(b) Validation Accuracy Comparison between different optimizers and Learning rates

(c) Training Loss Comparison between different optimizers and Learning rates



(d) Validation Loss Comparison between different optimizers and Learning rates

Figure 3: Training and Validation Accuracy and losses of different optimizers and learning rates with Bidirectional GRU

2.5 Analysis Table

Table 1 summarizes the final train and test accuracies achieved by each configuration after five epochs.

Optimizer	Bidirectional RNN	Learning rate	Train Accuracy	Test Accuracy
Adam	GRU	0.0001	87.77%	86.88%
Adam	LSTM	0.0001	87.88%	86.17%
Adam	GRU	0.001	76.29%	75.09%
Adam	GRU	0.05	0.00%	0.00%
SGD	GRU	0.001	79.79%	80.13%
SGD	GRU	0.05	0.00%	0.00%

Table 1: Training and testing accuracy for different optimizers, RNN types, and learning rates

2.6 Analysis

GRU with adam optimizer and learning rate of 0.0001 was the best configuration, achieving high training and test accuracy with good generalization. The choice of a lower learning rate was crucial for stable training, while the use of GRU provided a good balance between model complexity and effective sequence learning.

- **GRU (Adam Optimizer, Learning rate=0.0001)**: This configuration achieved high training and test accuracy, indicating good generalization. The use of a small learning rate (0.0001) enabled stable weight updates, allowing the model to converge effectively without overshooting. The GRU's gating mechanism appears to have handled the sequential dependencies well.
- **LSTM (Adam optimizer, Learning rate=0.0001)**: The LSTM model performed similarly to GRU, with slightly higher training accuracy but slightly lower test accuracy. This could indicate mild overfitting, as the model may have learned the training data better but at the expense of generalization.

Regarding Optimizers and learning rates with GRU:

- **Adam Optimizer (lr = 0.001)**: Increasing the learning rate to 0.001 resulted in a decrease in both training and test accuracy. This suggests that the higher learning rate might have caused the model to make larger weight updates, leading to instability in the learning process. The model may have struggled to converge effectively within the five epochs
- **SGD (lr = 0.001)**: Using SGD with a learning rate of 0.001 resulted in moderate training and test accuracy. This indicates that SGD, though less adaptive than Adam, was able to learn effectively at this learning rate. The training and test accuracies are quite similar, indicating good generalization.
- **SGD, Adam (lr = 0.05)**: Both models failed to train at all with a learning rate of 0.05. This high learning rate likely caused exploding gradients, leading models to diverge. The weight updates were likely too large, preventing the model from learning anything meaningful.

3 Task 2: Fine-Tuning Pre-trained ResNet50 Model

3.1 Fine-Tuning Strategy and analysis

In Task 2, we explored the potential of transfer learning by fine-tuning a pre-trained ResNet50 model for text recognition. The pre-trained ResNet50 was modified by removing its fully connected layers, and custom fully connected layers were added to adapt it to the sequence recognition task. Below is an analysis of the approach and the observed results:

a. **Slower convergence during fine tuning**: The fine-tuning process took many epochs to achieve around 70% validation accuracy. This suggests that the initial layers, which were pre-trained on ImageNet, provided general features, but adapting these features to the text sequence recognition task was challenging. The slow convergence indicates that the pre-trained features from natural images did not directly translate to text recognition without significant adaptation. As a result, the model needed many updates to learn text-specific features, even though pre-trained weights were used.

b. **Challenges with Transfer Learning**: The task required modifying the final layers of the pre-trained ResNet50 to output a sequence suitable for text recognition. However, because text features are quite different from the natural image features that ResNet50 was trained on, a substantial number of epochs were required to properly tune these weights for the new domain. This highlights a key challenge of transfer learning: although pre-trained models can speed up training and reduce data requirements, they may still need a significant amount of fine-tuning for highly specialized tasks, such as text recognition from complex backgrounds.

c. **Freezing and Unfreezing Layers**: The initial convolutional layers were frozen to preserve general low-level features like edges and textures, which are still useful in text images. Only higher-level layers were unfrozen and retrained to adapt to text features.

d. **Model performance**: The training accuracy increased steadily over the epochs, but validation accuracy lagged behind, indicating potential overfitting. The model was able to memorize the training data to some extent, but generalization to the validation set was slower. The final validation accuracy was around 70%, which, while an improvement over training from scratch, still required significant computation and training time to reach. This shows that the fine-tuning approach improved performance, but the model needed substantial retraining to learn domain-specific features effectively.

This analysis helps understand the limitations of using a pre-trained model in a domain with specialized requirements and highlights the importance of adapting transfer learning strategies based on the specific task at hand.

3.2 Experimental Setup

Removed CNN layers from Task 1, added Resnet model and used LSTM with Adam optimizer and a learning rate of 0.0001 to do the text recognition.

- **Hardware**: Experiments were conducted on a NVIDIA T4 GPU to expedite training.
- **Evaluation Metrics**: Train accuracy, Validation accuracy, Train and Validation loss per epoch.

3.3 Results

The fine-tuned model demonstrated varying degrees of improvement in test accuracies, as depicted in Figure 4. And we can also see how the model predicted sample words from the dataset namely "Quint", "silicate", "Gossamer".

```
Epoch 9, Training Loss: 0.1701, Validation Loss: 0.8800, Train Char Acc: 0.8608, Val Char Acc: 0.6907

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: bilicots, Target: silicate
Prediction: Gossamen, Target: Gossamer
Training...
Epoch 10, Training Loss: 0.1483, Validation Loss: 0.8857, Train Char Acc: 0.8740, Val Char Acc: 0.6972

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: Gilicats, Target: silicate
Prediction: Gossemer, Target: Gossamer
Training...
Epoch 11, Training Loss: 0.1288, Validation Loss: 0.9389, Train Char Acc: 0.8854, Val Char Acc: 0.6986

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: gilioote, Target: silicate
Prediction: Gossemen, Target: Gossamer
Training...
Epoch 12, Training Loss: 0.1143, Validation Loss: 0.9454, Train Char Acc: 0.8952, Val Char Acc: 0.6990

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: dilioate, Target: silicate
Prediction: Gossemer, Target: Gossamer
Training...
Epoch 13, Training Loss: 0.1034, Validation Loss: 0.9370, Train Char Acc: 0.9030, Val Char Acc: 0.6999

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: Gilicate, Target: silicate
Prediction: Gossemer, Target: Gossamer
Training...
Epoch 14, Training Loss: 0.0921, Validation Loss: 0.9863, Train Char Acc: 0.9106, Val Char Acc: 0.7062

Sample Predictions on Validation Set:
Prediction: QUINT, Target: QUINT
Prediction: dilioots, Target: silicate
Prediction: Gossamen, Target: Gossamer
```

Figure 4: Final Test Accuracies of Fine-Tuned Pre-trained Models

4 Task 3: Handwritten Words Recognition from Video

4.1 Methodology

In Task 3, the goal was to extend the text recognition capability of the trained CRNNResNet model to video data by integrating OpenCV for video processing. The process involved several stages, including frame extraction, text detection, preprocessing, and prediction using the trained model.

1. **Video frame Extraction**: The video was processed frame-by-frame using OpenCV's VideoCapture. Each frame was read sequentially, and a limit of 100 frames was set for demonstration purposes in Colab. The video file was opened, and each frame was extracted to analyze the content and identify potential text regions.

2. **Text Detection Using Contour Detection**: Each frame was converted to grayscale to simplify processing. A Gaussian blur was applied to reduce noise and improve the accuracy of text detection. Adaptive thresholding was used to generate a binary image that highlights text regions. Morphological dilation was applied to connect adjacent letters into larger, word-level segments using a larger kernel. Contours were then identified from the processed binary image, and bounding boxes were calculated for each detected text region.

3. **Text Recognition**: The region was extracted from the original frame as a Region of Interest (ROI). The ROI was then converted from BGR to RGB and transformed into a PIL Image to be processed using PyTorch's transforms module. Resize to 224x224 pixels to match the input size requirement of the model. Normalization based on ImageNet mean and standard deviation values. The transformed image was passed through the fine-tuned CRNNResNet model to predict the sequence of characters.

4. **Prediction Decoding**: The output from the model was decoded using a custom function (decodepredictionscrnn). The model output was converted to probabilities using the softmax function and the most probable character was selected for each time step. The sequence of predicted indices was then mapped back to characters using the character-to-index dictionary (idxtochar). Consecutive duplicate predictions and ¡BLANK¿ labels were removed to produce a meaningful text sequence.

5. **Displaying Results**: For each detected text box, the predicted text was drawn on the original frame. The bounding box and the corresponding text prediction were drawn using OpenCV functions (cv2.rectangle and cv2.putText). The annotated frame was then displayed using cv2imshow to visualize the real-time text recognition process.

A snippet of a function to detect text regions using improved contours is shown below to illustrate how contours are identified and annotated in video frames:

```
def get_text_contours(frame):
    # Convert to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    # Apply adaptive thresholding
    thresh = cv2.adaptiveThreshold(blurred, 255, cv2.
       ADAPTIVE_THRESH_GAUSSIAN_C,
                                   cv2.THRESH_BINARY_INV, 15, 10)

    # Apply dilation to connect adjacent letters
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (15, 15))
    dilated = cv2.dilate(thresh, kernel, iterations=1)

    # Find contours on the dilated image
    contours, hierarchy = cv2.findContours(dilated, cv2.
       RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # List to hold bounding boxes of detected text regions
    bounding_boxes = []

    for cnt in contours:
        # Calculate bounding box for each contour
        x, y, w, h = cv2.boundingRect(cnt)
        # Filter out small or large regions to focus on word-
           level contours
```

```
        if w > 50 and h > 20:  # Adjust these thresholds based on
            your video
            bounding_boxes.append((x, y, x + w, y + h))

    return bounding_boxes
```

Listing 2: Character Detection

4.2 Frame Processing and Recognition

Each selected frame was processed to enhance text visibility and prepare it for recognition. Figure 5 illustrates recognized words in a hand written text video. Wrote HUMAN COMPUTER INTERACTION in a paper and took a video and is used for the character recognition task.

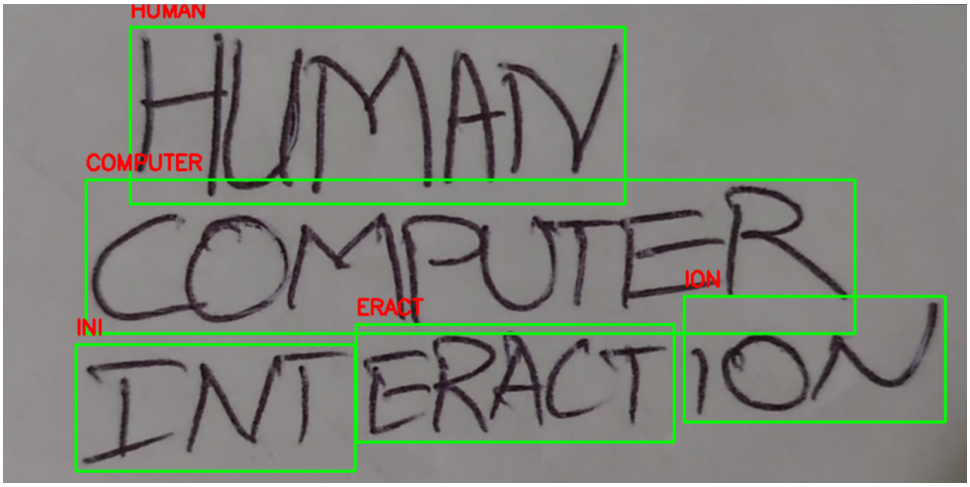


Figure 5: Text detection

4.3 Results

The system successfully recognized handwritten alphabets across all selected frames, demonstrating robustness in dynamic scenarios. An example frame is shown in the Fig 5

4.4 Challenges and Solutions

- **Text Detection in Complex Backgrounds**: In video frames, text is often embedded in complex backgrounds with varying colors, textures, and lighting conditions, which makes accurate text detection difficult. Adaptive thresholding was used to dynamically binarize the grayscale image based on local pixel intensities, which helps in isolating text even when there is a non-uniform background. Morphological dilation was applied to better connect fragmented parts of the text and group individual letters into cohesive word regions. This helped improve the overall text detection quality.
- **Variations in Text Size and Font**: Text in the video could vary significantly in size, font, and aspect ratio, making it challenging to design a detection method that

generalizes well to all variations. A multi-stage filtering approach was implemented to filter out detected regions that did not match the typical size of text, reducing false positives

- **Motion Blur in Video Frames**: Motion blur caused by either camera movement or object movement within the video makes it difficult to extract clear, readable text. Gaussian blur was initially applied to reduce noise before thresholding. To further enhance robustness to motion blur, a higher quality thresholding method, like adaptive Gaussian thresholding, was used to dynamically adjust to the blurriness of different regions in the frame.

5 Discussion

The goal of this project was to develop and evaluate a model capable of recognizing text in images and videos, involving a progression from training custom architectures, to transfer learning with pre-trained models, and extending the capabilities to real-time video text recognition. Each task offered unique insights into the effectiveness of different modeling and preprocessing strategies for text recognition.

- **Task 1** involved the design and training of a CRNN architecture, leveraging both LSTM and GRU units. The experiments showed that careful selection of learning rates was crucial. The Adam optimizer with a small learning rate consistently yielded the best results, highlighting the sensitivity of the training process to hyperparameter tuning. The GRU-based model demonstrated slightly better generalization compared to LSTM, which may be attributed to its relatively simpler structure that prevented overfitting.

- **Task 2** focused on transfer learning by fine-tuning a pre-trained ResNet50 model. The results highlighted the advantages of leveraging pre-trained networks, especially when faced with limited training data and computational resources. However, even with transfer learning, achieving decent validation accuracy required many epochs, indicating that the features learned by ResNet50 on natural images needed significant adaptation for text recognition. This demonstrated the challenges of applying pre-trained models to new domains, where specialized features are required.

- **Task 3** extended the trained model to perform video text recognition using OpenCV. This task highlighted new challenges, such as dealing with motion blur, variable lighting conditions, and real-time constraints. The approach of using adaptive thresholding and morphological operations for text detection worked well for simpler frames, but it struggled with more complex backgrounds or frames with significant distortion. The challenges in accurately detecting and recognizing text in dynamic video environments emphasized the importance of robust preprocessing methods.

Throughout the project, several challenges were identified, such as overfitting, slow convergence, inconsistent text detection, and resource limitations. Solutions such as using adaptive preprocessing techniques, regularization, and transfer learning helped mitigate these issues, but further improvements are still possible. For example, integrating a more advanced text detection method (e.g., EAST detector) and employing language models for post-processing could improve both detection accuracy and recognition quality.

6 Conclusion

The project successfully developed a pipeline for recognizing text from both images and video sequences using a combination of CRNN architectures, transfer learning, and OpenCV for real-time video processing. The results showed that custom architectures, pre-trained models, and effective preprocessing can all contribute significantly to text recognition performance. Despite the successes, challenges remain in adapting these models for more generalized and real-time use cases. Future work could include integrating deep learning-based text detectors, using Transformer-based architectures for sequence recognition, and exploring real-time optimizations such as model quantization to handle dynamic content more effectively

7 Future Work

To further enhance the performance and capabilities of the developed pipeline, the following improvements are suggested:

- **Advanced Text Detection:** Replace contour-based detection with deep learning-based methods like EAST or CRAFT for more accurate detection under varying conditions.
- **Language Modeling for Prediction Refinement:** Integrate a character-level language model to refine the recognition output based on context, reducing recognition errors.
- **Handling Real-Time Constraints:** Apply model compression techniques, such as quantization or distillation, to make the model more efficient for real-time video processing tasks.
- **Dataset Expansion:** Due to computational constraints we trained the data on 300,000 images. Training and Testing the data on whole set would make the model learn effectively and perform better on unseen data.