A

MINI PROJECT REPORT

ON

CAPTCHA SOLVER

Submitted in partial fulfillment of the requirements
For the award of Degree of

BACHELOR OF ENGINEERING

IN

CSE (Artificial Intelligence & Machine Learning)

Submitted By

Ankam Veena Sree	245322748004
Kallagunta Mythili	245322748033
Kathoju Tejashwini	245322748035

Under the guidance

Of

Mrs M. Deepika

Assistant Professor

Department of CSE(AIML)

NEIL GOGTE INSTITUTE OF TECHNOLOGY
Kachavanisingaram Village, Hyderabad, Telangana 500058.

February 2025

NEIL GOGTE INSTITUTE OF TECHNOLOGY

A Unit of Keshav Memorial Technical Education (KMTES)

Approved by AICTE, New Delhi & Affiliated to Osmania University, Hyderabad

CERTIFICATE

This is to certify that the project work (PW533CSM) entitled "**CAPTCHA SOLVER**" *is a bonafide work carried out by* **A. Veena Sree (245322748004), K. Mythili (245322748033), K. Tejashwini (245322748035)** of III-year V semester **Bachelor of Engineering** *in* **CSE (Artificial Intelligence & Machine Learning)** *by Osmania University, Hyderabad during the academic year* **2024-2025** *is a record of bonafide work carried out by them. The results embodied in this report have not been submitted to any other University or Institution for the award of any degree*

Internal Guide	**Head of Department**
Mrs M. Deepika	Dr.K. Vinuthna Reddy
Assistant Professor	Associate Professor

External

NEIL GOGTE INSTITUTE OF TECHNOLOGY

A Unit of Keshav Memorial Technical Education (KMTES) Approved by AICTE, New Delhi & Affiliated to Osmania University, Hyderabad

DECLARATION

We hereby declare that the Mini Project Report entitled, "**CAPTCHA SOLVER**" submitted for the B.E degree is entirely our work and all ideas and references have been duly acknowledged. It does not contain any work for the award of any other degree.

Date:

Ankam Veena Sree	245322748004
Kallagunta Mythili	245322748033
Kathoju Tejashwini	245322748035

ACKNOWLEDGEMENT

We are happy to express our deep sense of gratitude to the principal of the college **Dr. R. Shyam Sunder, Professor**, Neil Gogte Institute of Technology, for having provided us with adequate facilities to pursue our project.

We would like to thank**, Dr. K. Vinuthna Reddy**, **Head of the Department, CSE (Artificial Intelligence & Machine Learning)** Neil Gogte Institute of Technology, for having provided the freedom to use all the facilities available in the department, especially the laboratories and the library.

We would also like to thank my internal guide **Mrs M. Deepika Asst. Professor** for her technical guidance & constant encouragement.

We sincerely thank our seniors and all the teaching and non-teaching staff of the Department of CSE (Artificial Intelligence & Machine Learning) and Information Technology for their timely suggestions, healthy criticism and motivation during the course of this work.

Finally, we express our immense gratitude with pleasure to the other individuals who have either directly or indirectly contributed to our need at the right time for the development and success of this work.

ABSTRACT

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) systems are widely used to prevent automated bots from accessing web services by presenting challenges that are easy for humans but difficult for machines. This project focuses on solving CAPTCHA challenges using Optical Character Recognition (OCR) techniques based on Convolutional Neural Networks (CNNs), aimed at enhancing accuracy and reducing the need for manual intervention.

Traditional CAPTCHA-solving systems primarily use basic OCR techniques or simpler CNN models. These systems often rely on manual preprocessing, such as segmenting CAPTCHA images into individual characters. However, this approach struggles with more complex CAPTCHAs that feature overlapping, distorted, or noisy characters. The reliance on segmentation and the limited ability of these models to generalize across different CAPTCHA formats results in reduced accuracy and efficiency.

The proposed system overcomes these limitations by utilizing an advanced CNN combined with Connectionist Temporal Classification (CTC) loss. This eliminates the need for manual segmentation, allowing the model to process the entire CAPTCHA image and predict character sequences directly. This approach improves accuracy and generalization, making it effective across various CAPTCHA formats.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER-1

INTRODUCTION

1.1 PROBLEM STATEMENT

CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) are a widely used mechanism to distinguish between human users and automated bots. They often appear in login forms, online transactions, and other scenarios requiring user authentication. CAPTCHAs use distorted alphanumeric characters, noise, and background patterns to make automated solving challenging.

However, as essential as they are for security, CAPTCHAs introduce usability issues, especially for people with disabilities and for tasks that require repetitive manual solving. For instance, organizations and developers often face challenges when testing websites or automating workflows that involve CAPTCHA-protected endpoints.

This project seeks to address this problem by building an automated CAPTCHA-solving system using Optical Character Recognition (OCR) techniques. The proposed solution uses Convolutional Neural Networks (CNNs) to predict alphanumeric sequences from CAPTCHA images. The system aims to bypass the inefficiency of manual CAPTCHA solving while maintaining high accuracy and robustness against noise, distortions, and font variations.

1.2 MOTIVATION

The motivation for this project stems from both the limitations of current CAPTCHA-solving methods and the opportunities presented by advancements in machine learning and computer vision.

1. **Improving Usability:** CAPTCHAs can be frustrating for users, particularly in scenarios where they must solve multiple CAPTCHAs repeatedly. Automating this process can improve user experience and save time.

2. **Accessibility:** People with visual impairments or cognitive difficulties often struggle with CAPTCHAs, which can lead to exclusion. An automated CAPTCHA solver could enhance accessibility for such users.

3. **Advances in Technology:** Recent developments in deep learning, particularly CNNs, have significantly improved the accuracy of OCR systems. Leveraging these techniques for CAPTCHA solving provides an opportunity to push the boundaries of what OCR models can achieve in highly distorted and noisy data environments.

4. **Applications:** Automated CAPTCHA solvers are useful in fields such as automated testing, web scraping, and system penetration testing. These solvers can streamline processes where CAPTCHA-protected data or services need to be accessed efficiently.

This project aims to bridge the gap between human-level CAPTCHA-solving accuracy and the capabilities of machine learning models while ensuring robustness and efficiency.

1.3 SCOPE

The scope of this project is well-defined, focusing on CAPTCHA-solving through deep learning techniques. Key aspects include:

1. **Data Collection and Preprocessing:**

   - Working with alphanumeric CAPTCHA datasets. o Preprocessing includes handling distortions, segmenting characters if necessary, and normalizing data for model training.

2. **Model Design and Training:**

   - Building a CNN-based OCR model optimized for recognizing distorted alphanumeric sequences.
   - Using appropriate loss functions and evaluation metrics to ensure high accuracy.

3. **Testing and Evaluation:**

- Evaluating the model on unseen CAPTCHA images to test robustness against variations in fonts, noise, and distortions. o Comparing performance metrics such as accuracy, precision, recall, and inference time.

4. **Applications:**

   - The final model can be applied to automate CAPTCHA-solving in testing environments, improving accessibility, and reducing manual effort.

Exclusions: The project does not focus on breaking CAPTCHAs maliciously or bypassing security for unethical purposes. Its purpose is strictly educational and for automation in ethical use cases.

1.4 OUTLINE

The project report is organized as follows:

1. **Introduction**

   - Covers the problem statement, motivation, and scope of the CAPTCHA solver project.

2. **Literature Survey**

   - Explores existing CAPTCHA-solving techniques, challenges in CAPTCHA recognition, and the role of deep learning, particularly CNNs, in OCR tasks.
   - Reviews previous models and tools used in CAPTCHA-solving.

3. **Implementation**

   - Explains the technical aspects of building and implementing the OCR model using CNNs.
   - Covers the tools, libraries, and frameworks used, such as TensorFlow or PyTorch.

4. **Conclusion and Future Work**

   - Summarizes the achievements of the project and its significance.

- Explores potential improvements, such as expanding the dataset, using transformerbased models (e.g., Vision Transformers), or exploring adversarial training to handle more complex CAPTCHAs.

CHAPTER-2

LITERATURE SURVEY

2.1 EXISTING SYSTEM

CAPTCHA Recognition Using Deep Learning Models

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) systems are widely used to distinguish human users from bots on websites. Text-based CAPTCHAs, involving distorted characters and background noise, have become increasingly vulnerable to attacks from deep learning models. The use of Convolutional Neural Networks (CNNs) has been a common approach for CAPTCHA recognition due to their ability to extract spatial features. However, the complex noise and interference found in CAPTCHAs often degrade the performance of CNN-based models.

To address these challenges, recent research has integrated Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks into CAPTCHA recognition systems. These models are effective at capturing the sequential relationships between characters, improving recognition accuracy when characters are deformed or connected. The Convolutional Recurrent Neural Network (CRNN) model, which combines CNNs for feature extraction with RNNs/LSTMs for sequence learning, has shown promising results in recognizing distorted CAPTCHA images.

Moreover, attention mechanisms such as the Convolutional Block Attention Module (CBAM) have been introduced to allow models to focus on critical parts of the CAPTCHA image, further improving recognition accuracy. These mechanisms enable the model to handle challenging CAPTCHA images by emphasizing relevant character features, even in the presence of noise and interference.

Handling noise and interference, which are deliberately added in CAPTCHAs to thwart automated systems, remains a significant challenge. Researchers have introduced Adaptive Fusion Filter Networks (AFFN) to combat these issues. These filters help reduce noise while preserving important features, improving the accuracy of CAPTCHA recognition models without significantly increasing computational complexity.

Additionally, some researchers have explored the use of Generative Adversarial Networks (GANs) to generate synthetic CAPTCHAs for training purposes, enhancing the robustness of models by exposing them to a variety of CAPTCHA styles. However, GAN-based methods are computationally expensive, which limits their practical application in small-scale CAPTCHA recognition tasks.

In conclusion, deep learning methods, particularly those combining CNNs, RNNs/LSTMs, and attention mechanisms, are at the forefront of CAPTCHA recognition research. These approaches offer a good balance of accuracy and computational efficiency, overcoming the limitations of traditional machine learning-based methods and making CAPTCHA recognition more robust to noise, distortion, and interference.

This survey highlights the recent advancements in CAPTCHA recognition, focusing on the use of deep learning models, adaptive filtering, and attention mechanisms, which are relevant to your CAPTCHA solver project.

2.2 PROPOSED SYSTEM

CAPTCHA Solver Using Layout-Aware OCR and Deep Learning :

CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) are widely used to prevent automated bots from accessing websites and services. They often involve distorted images containing alphanumeric characters, which are challenging for machines to decode but relatively easy for humans. The challenge lies in designing systems that can accurately interpret these distorted characters while maintaining robustness to adversarial variations, such as noise, background patterns, and distortions. Over the years, various approaches have been explored to break CAPTCHAs, with deep learning techniques emerging as the most promising solutions.

Traditional Methods: Early approaches to CAPTCHA solving focused on rule-based methods such as edge detection, thresholding, and character segmentation. These techniques were limited by their inability to handle complex distortion patterns and overlapping characters. Moreover, they required extensive hand-crafted features, which made them rigid and inefficient for handling the diversity of CAPTCHAs used across different websites.

Machine Learning Approaches: As machine learning techniques advanced, researchers began using models such as Support Vector Machines (SVMs) and Random Forests to tackle CAPTCHA problems. These methods typically involved feature extraction and pattern recognition techniques to classify characters. While these approaches showed improvement over traditional methods, they still struggled with complex CAPTCHAs that incorporated noise and varying font styles. The need for more flexible, end-to-end models led to the adoption of deep learning.

Deep Learning Models: The advent of Convolutional Neural Networks (CNNs) revolutionized the field of image recognition, enabling the automatic extraction of features and achieving state-ofthe-art results on various image classification tasks. Recent studies have applied CNNs to CAPTCHA solving, where the model directly learns the features from the raw image pixels. However, these models still face challenges when dealing with CAPTCHAs that include multicharacter strings, complex layouts, or highly distorted images.

Connectionist Temporal Classification (CTC): To handle the sequence nature of CAPTCHA recognition, many deep learning-based approaches employ Connectionist Temporal Classification (CTC) loss, which is particularly useful when there is no explicit alignment between the input (image) and output (characters). CTC enables the model to output sequences of variable lengths, making it ideal for tasks like CAPTCHA solving. CTC-based models have been successfully employed in a wide range of text recognition tasks, including handwriting recognition and speechto-text conversion, demonstrating their effectiveness in CAPTCHA challenges.

Layout-Aware OCR: One notable advancement is the integration of Layout-Aware Optical Character Recognition (OCR) techniques. These methods aim to understand the spatial relationships between characters in an image, which is crucial for solving CAPTCHAs with complex layouts or overlapping characters. LayoutLM, a model designed for document understanding, has been successfully applied to various tasks requiring a joint understanding of text and layout. This approach provides a more robust solution for CAPTCHA recognition, especially in cases where traditional OCR models fail to extract text accurately due to cluttered or complex backgrounds.

Recent Advances and Future Directions: Recent advancements in Vision-Language Models (VLMs) and Transformer-based architectures, such as LayoutLM and Vision Transformers (ViT), have shown significant promise in improving the accuracy of CAPTCHA solvers. These models leverage both visual and textual information, making them ideal for handling complex CAPTCHAs that involve intricate layouts and diverse fonts. Additionally, techniques such as attention mechanisms and beam search decoding have further improved the performance of deep learning-based CAPTCHA solvers by better handling ambiguities in the predicted text sequences.

As CAPTCHA systems continue to evolve with increasingly sophisticated distortions and designs, there is a growing need for more advanced models that can adapt to these challenges. Future work in this field may focus on improving the generalization of CAPTCHA solvers by utilizing largescale datasets, integrating adversarial training to handle evolving CAPTCHA techniques, and developing more efficient inference methods for real-time applications.

CHAPTER-3

SOFTWARE REQUIREMENTS SPECIFICATION

3.1. Overall Description

This SRS outlines the comprehensive framework for the CAPTCHA Solver project. The purpose of this document is to provide a detailed description of the system, including its features, user interfaces, operational constraints, and expected responses to various external inputs. It serves as a guide for both stakeholders and developers involved in the project.

3.2. Operating Environment

Software Requirements

Operating System	:	Windows 10, Linux (Ubuntu 20.04)
Front End	:	HTML, CSS, JavaScript
Back End	:	Python
Database	:	SQLite
Server	:	Flask (Web Framework)
Development Kit	:	Visual Studio Code (Latest Version)

Hardware Requirements

Processor	:	Intel Core i5 (Min)
Speed	:	2.5 GHz (Min)
RAM	:	8 GB (Min)
Hard Disk	:	10 GB (Min)

3.3. Functional Requirements

Input Handling:

The system shall accept image data of CAPTCHA formats (e.g., alphanumeric).

OCR Integration:

The system shall implement Optical Character Recognition (OCR) to analyze and extract text from CAPTCHA images.

User Feedback:

Users shall receive immediate feedback on the success or failure of the CAPTCHA solving attempts.

Logging:

The system shall log all CAPTCHA solving attempts, including timestamps and results.

Performance Metrics:

The system shall track accuracy rates and response times for CAPTCHA solving.

3.4. Non-Functional Requirements

3.4.1 Performance Requirements

Performance requirements pertain to the expected operational metrics of the CAPTCHA Solver.

Response Time:

The average response time for solving a CAPTCHA shall be less than 5 seconds.

Recovery Time:

In the event of a system failure, recovery procedures shall restore operations within 15 seconds. Average repair time shall be less than 30 minutes.

Start-Up/Shutdown Time:

The system shall be operational within 2 minutes of startup.

Capacity:

The system shall handle up to 1000 concurrent users without degradation in performance.

Utilization of Resources:

The system shall maintain a database size limit of 500,000 CAPTCHA records. If this limit is exceeded, older records shall be archived and removed from active storage.

3.4.2 Safety Requirements

-NA-

3.4.3 Security Requirements

The system shall implement user authentication, ensuring that only registered users can access the CAPTCHA solving functionality. Additionally, it shall employ secure methods for handling image data to prevent unauthorized access.

3.4.4 Software Quality Attributes

Reliability:

The system shall have failover mechanisms, ensuring that if the primary server crashes, a backup server can take over seamlessly.

Availability:

The system will be available 24/7, allowing users to access the CAPTCHA solving service at any time.

Security:

Each user will be assigned a unique registered ID to enhance accountability and security.

Maintainability:

The system will be designed for ease of maintenance with modular architecture, extensive documentation, and adherence to coding standards.

Usability:

The user interfaces will be intuitive and user-friendly, allowing users of varying technical backgrounds to easily interact with the system.

Scalability:

The system will be designed to accommodate future enhancements, including support for additional CAPTCHA types or improved OCR algorithms, without requiring extensive redesign.

CHAPTER-4

SYSTEM DESIGN

4.1 UML DIAGRAMS

4.1.1 Use case Diagram:

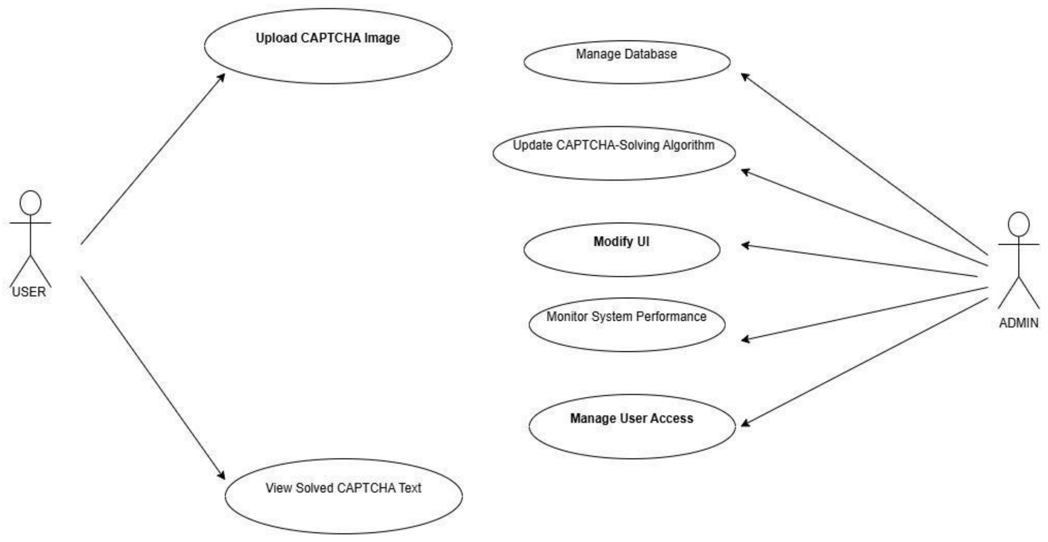

Fig 4.1: Use Case Diagram

Description:

- User uploads CAPTCHA images for solving by the system.
- User views solved CAPTCHA text after processing.
- Admin manages the database for data storage and retrieval.
- Admin updates CAPTCHA-solving algorithms to improve accuracy and efficiency.
- Admin modifies the UI to enhance user experience.
- Admin monitors system performance to ensure smooth operation.

4.1.2 Class Diagram:

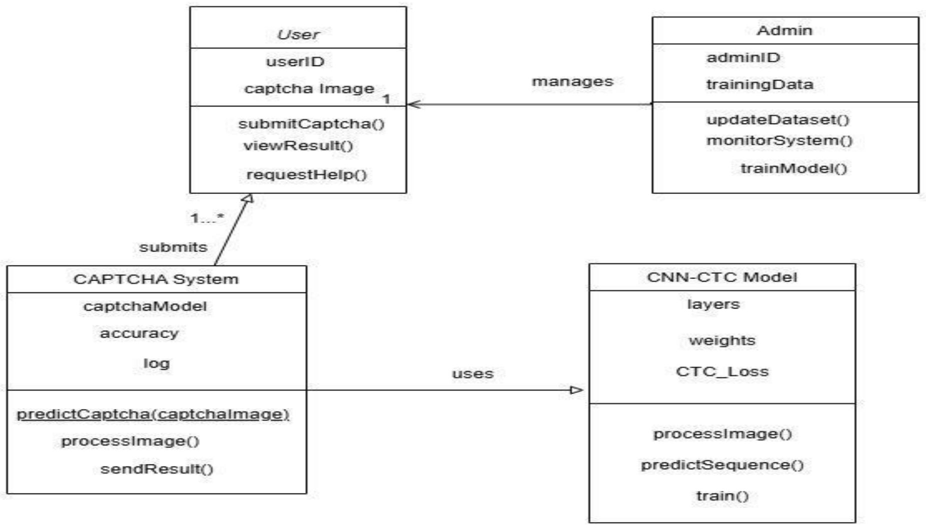

Fig 4.2: Class diagram for Application.

Description:

1. **User class**: Manages userID and captchaImage with methods submitCaptcha(), viewResult(), and requestHelp().
2. **Admin class**: Handles adminID and trainingData with methods updateDataset(), monitorSystem(), and trainModel().
3. **CAPTCHA System class**: Uses attributes captchaModel, accuracy, and log with methods predictCaptcha(), processImage(), and sendResult().
4. **CNN-CTC Model class**: Defines layers, weights, and CTC_Loss with methods processImage(), predictSequence(), and train().
5. **User-CAPTCHA System relationship**: Users submit CAPTCHA images to the system (1-to-many relationship).
6. **Admin-CAPTCHA System relationship**: Admins maintains the CAPTCHA system.
7. **CAPTCHA System-CNN-CTC Model relationship**: The CAPTCHA system uses the CNN-CTC model for CAPTCHA solving and training.

4.1.3 Sequence Diagram:

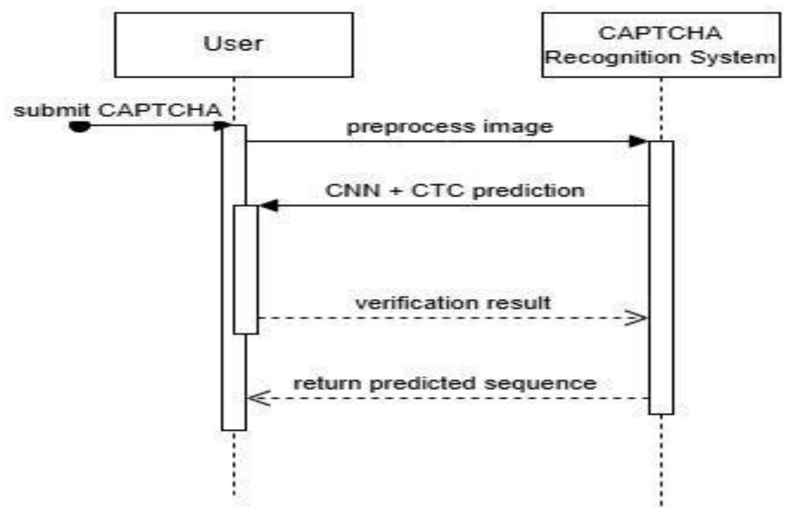

Fig 4.3: Sequence Diagram for application.

Description:

- **User submits CAPTCHA:** The user submits a CAPTCHA image to the recognition system.
- **System preprocesses image:** The CAPTCHA recognition system preprocesses the submitted image.
- **System performs CNN + CTC prediction:** The system applies the CNN model and CTC algorithm to predict the text.
- **Verification result is sent:** The system verifies the predicted sequence and prepares the result.
- **Predicted sequence is returned:** The recognition system sends the predicted CAPTCHA text back to the user.

CHAPTER-5

IMPLEMENTATION

```
#importing required libraries import os
os.environ["KERAS_BACKEND"] = "tensorflow"
import numpy as np import matplotlib.pyplot as
plt from pathlib import Path import tensorflow as
tf import keras from keras import ops from keras
import layers #count of images # Path to the data
directory data_dir = Path("./captcha_images_v2/")
# Get list of all the images images = sorted(list(map(str,
list(data_dir.glob("*.png"))))) labels = [img.split(os.path.sep)[-
1].split(".png")[0] for img in images] characters = set(char for label
in labels for char in label) characters = sorted(list(characters))
print("Number of images found: ", len(images)) print("Number of
labels found: ", len(labels))
print("Number of unique characters: ", len(characters))
print("Characters present: ", characters) # Batch size
for training and validation batch_size = 16
# Desired image dimensions
img_width = 200 img_height
= 50 downsample_factor = 4
```

```
# Maximum length of any captcha in the dataset max_length
= max([len(label) for label in labels])
# Mapping characters to integers char_to_num =
layers.StringLookup(vocabulary=list(characters), mask_token=None)
# Mapping integers back to original characters num_to_char =
layers.StringLookup(     vocabulary=char_to_num.get_vocabulary(),
mask_token=None, invert=True
)
def split_data(images, labels, train_size=0.9, shuffle=True):
   # 1. Get the total size of the dataset
size = len(images)
   # 2. Make an indices array and shuffle it, if required
   indices = ops.arange(size) if shuffle:
       indices = keras.random.shuffle(indices) #
    3. Get the size of training samples
    train_samples = int(size * train_size)
   # 4. Split data into training and validation sets     x_train, y_train =
images[indices[:train_samples]], labels[indices[:train_samples]]     x_valid, y_valid =
images[indices[train_samples:]], labels[indices[train_samples:]]     return x_train,
x_valid, y_train, y_valid
```

```
# Splitting data into training and validation sets x_train, x_valid, y_train, y_valid
= split_data(np.array(images), np.array(labels)) def
encode_single_sample(img_path, label):
    # 1. Read image     img = tf.io.read_file(img_path)
# 2. Decode and convert to grayscale     img =
tf.io.decode_png(img, channels=1)     # 3. Convert to
float32 in [0, 1] range     img =
tf.image.convert_image_dtype(img, tf.float32)
    # 4. Resize to the desired size     img =
ops.image.resize(img, [img_height, img_width])
    # 5. Transpose the image because we want the time #
    dimension to correspond to the width of the image.
    img = ops.transpose(img, axes=[1, 0, 2])
    # 6. Map the characters in label to numbers label =
    char_to_num(tf.strings.unicode_split(label, input_encoding="UTF-8"))
    # 7. Return a dict as our model is expecting two inputs return
    {"image": img, "label": label}
#Training the model def ctc_decode(y_pred, input_length, greedy=True,
beam_width=100, top_paths=1):
    input_shape = ops.shape(y_pred)     num_samples, num_steps = input_shape[0],
input_shape[1]     y_pred = ops.log(ops.transpose(y_pred, axes=[1, 0, 2]) +
keras.backend.epsilon())     input_length = ops.cast(input_length, dtype="int32")

if greedy:

        (decoded, log_prob) = tf.nn.ctc_greedy_decoder(

inputs=y_pred, sequence_length=input_length

        )

else:

        (decoded, log_prob) = tf.compat.v1.nn.ctc_beam_search_decoder(

inputs=y_pred,             sequence_length=input_length,

beam_width=beam_width,             top_paths=top_paths,

        )

    decoded_dense = []

    for st in decoded:

        st = tf.SparseTensor(st.indices, st.values, (num_samples, num_steps))

    decoded_dense.append(tf.sparse.to_dense(sp_input=st, default_value=-1)) return

    (decoded_dense, log_prob)

# Get the prediction model by extracting layers till the output layer

prediction_model = keras.models.Model(     model.input[0],

model.get_layer(name="dense2").output

)

prediction_model.summary()

# A utility function to decode the output of the network def

decode_batch_predictions(pred):
   input_len = np.ones(pred.shape[0]) * pred.shape[1]

   # Use greedy search. For complex tasks, you can use beam search

results = ctc_decode(pred, input_length=input_len, greedy=True)[0][0][

      :, :max_length

   ]

   # Iterate over the results and get back the text

output_text = []     for res in results:

      res = tf.strings.reduce_join(num_to_char(res)).numpy().decode("utf-8")

   output_text.append(res) return output_text

# Let's check results on some validation samples for batch in

validation_dataset.take(1):     batch_images = batch["image"]     batch_labels =

batch["label"]     preds = prediction_model.predict(batch_images)     pred_texts

= decode_batch_predictions(preds)     orig_texts = []     for label in batch_labels:

label = tf.strings.reduce_join(num_to_char(label)).numpy().decode("utf-8")

orig_texts.append(label)

   _, ax = plt.subplots(4, 4, figsize=(15, 5))

for i in range(len(pred_texts)):

      img = (batch_images[i, :, :, 0] * 255).numpy().astype(np.uint8)

img = img.T         title = f"Prediction: {pred_texts[i]}"         ax[i //

4, i % 4].imshow(img, cmap="gray")         ax[i // 4, i %

4].set_title(title)         ax[i // 4, i % 4].axis("off") plt.show() #Front

end code import os import tensorflow as tf
```

```
from flask import Flask, request, jsonify, render_template

import numpy as np from tensorflow.keras.models import

load_model from tensorflow.keras.preprocessing import

image import matplotlib.pyplot as plt # Define the Flask

app app = Flask(__name__)

# Assuming CTCLayer is already defined as in the provided code

from tensorflow.keras import layers class

CTCLayer(layers.Layer):     def __init__(self, **kwargs):

        super(CTCLayer, self).__init__(**kwargs)

def call(self, y_true, y_pred):

        return tf.reduce_mean(tf.nn.ctc_loss(labels=y_true, logits=y_pred, label_length=None,
logit_length=None))     def compute_output_shape(self, input_shape):

        return (input_shape[0], 1)

# Load the trained model (ensure the custom CTCLayer is included) model =

load_model('model/my_model.h5', custom_objects={'CTCLayer': CTCLayer})

# Function to prepare a new image for prediction def

prepare_image(img_path, img_width=200, img_height=50):

    img = tf.io.read_file(img_path)

    img = tf.io.decode_png(img, channels=1)     img =

tf.image.convert_image_dtype(img, tf.float32)     img = tf.image.resize(img,

[img_height, img_width])     img = tf.transpose(img, perm=[1, 0, 2])     img
```

```
= tf.expand_dims(img, axis=0)     input_length = np.ones((img.shape[0], 1),

dtype=np.int32) * img.shape[1]     return img, input_length

# Function to decode the prediction output def

decode_prediction(prediction, num_to_char):

    input_length = np.ones(prediction.shape[0]) * prediction.shape[1]     decoded, _

= tf.keras.backend.ctc_decode(prediction, input_length, greedy=True)

decoded_text = []     for seq in decoded[0]:

        text = ''.join([num_to_char(char.numpy()).numpy().decode("utf-8") for char in seq if char !=
-1])

decoded_text.append(text)

return decoded_text[0]

# Utility function for mapping numbers to characters def

num_to_char(num_tensor):

                                                            char_map                                 =
"12345678bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

    char_map_tensor = tf.constant(list(char_map), dtype=tf.string)

num_tensor = tf.clip_by_value(num_tensor, 0, len(char_map) - 1) return

tf.gather(char_map_tensor, tf.cast(num_tensor, dtype=tf.int32))


# Routes

@app.route('/') def

home():

    return render_template('welcome.html')
```

```
@app.route('/main') def

main():

   return render_template('index.html')

# Endpoint for file upload and prediction

@app.route('/predict', methods=['POST']) def

predict_captcha():     if 'captcha_image' not in

request.files:         return jsonify({'error': 'No file

uploaded'}), 400     file =

request.files['captcha_image']     if file.filename ==

'':

       return jsonify({'error': 'No file selected'}), 400

# Save the uploaded image temporarily

upload_folder = 'uploads'     if not

os.path.exists(upload_folder):

       os.makedirs(upload_folder)     img_path =

os.path.join(upload_folder, file.filename)

file.save(img_path)

   # Prepare the image and make a prediction     img,

input_length = prepare_image(img_path)     prediction_model

= tf.keras.models.Model(         model.input[0],

model.get_layer(name="dense2").output)     prediction =
prediction_model.predict(img)     decoded_text =
decode_prediction(prediction, num_to_char)
    # Delete the uploaded image after prediction
os.remove(img_path)
    # Return the predicted text as a response
return decoded_text, 200
# @app.route('/welcome')
# def welcome():
#     return render_template('welcome.html')
# # Frontend form to upload an image
# @app.route('/')
# def main():
#     return render_template('index.html')
# @app.route('/') #
def home():
#     return welcome() if
__name__ == "__main__":
app.run(debug=True)
```

CHAPTER-6

TESTING

6.1 TEST CASES

Table 6.1.1: Test Case 1

Test Case 1 #ID	Test Case Type	Test Case Description	Expected Value	Actual Value	Result
TC_1	Model Accuracy	Captcha solver accurately predicts text from sample images	Predicted text matches ground truth	Predicted text is correct	PASS

Table 6.1.2: Test Case 2

Test Case 1 #ID	Test Case Type	Test Case Description	Expected Value	Actual Value	Result
TC_2	API Functionality	Captcha solver API returns results within a reasonable time	API response time < 2 seconds	API response time = 1.8s	PASS

CHAPTER-7

SCREENSHOTS

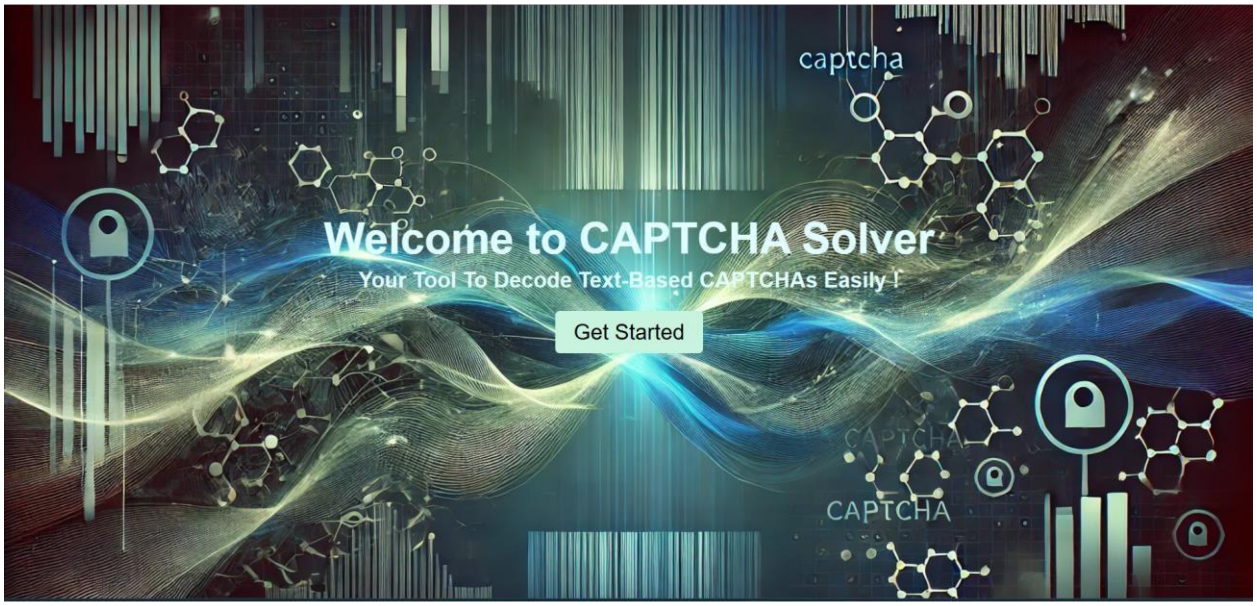
captcha
Welcome to CAPTCHA Solver
Your Tool To Decode Text-Based CAPTCHAs Easily !
Get Started
CAPTCHA
CAPTCHA

Fig 7.1 : Welcome page

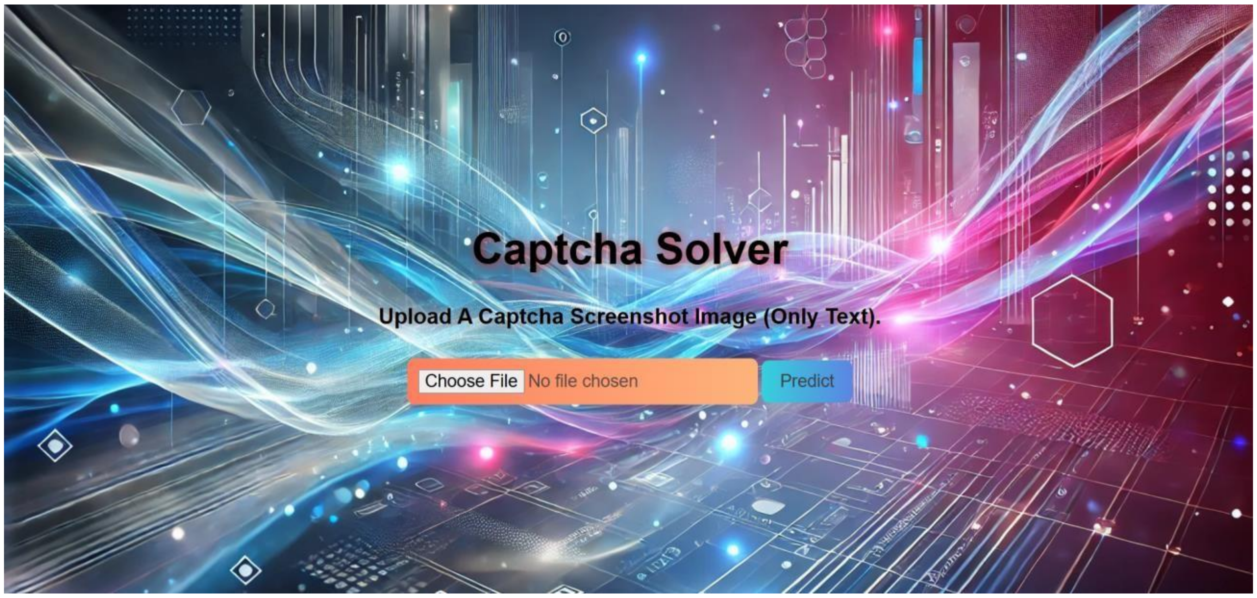

Fig 7.2 : Main page

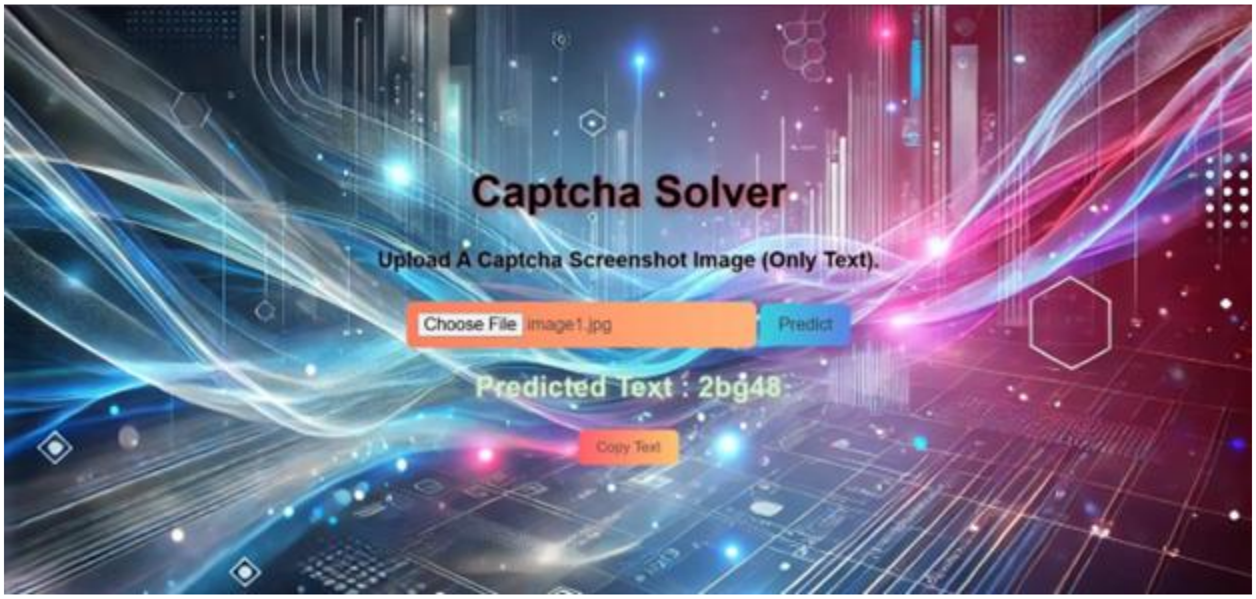

Fig 7.3 : Prediction page

CHAPTER-8

CONCLUSION AND FUTURE SCOPE

The CAPTCHA solver project successfully demonstrates the potential of deep learning techniques, particularly Convolutional Neural Networks (CNNs), in automating the solving of alphanumeric CAPTCHA challenges. By training the OCR model on a carefully curated dataset, the system was able to achieve robust performance in identifying and decoding distorted alphanumeric sequences from CAPTCHA images.

This project contributes to the field of automated CAPTCHA solving by reducing manual effort, improving accessibility, and showcasing how modern machine learning algorithms can tackle realworld problems. While the current system is limited to solving alphanumeric CAPTCHAs, it highlights the feasibility of further extending the capabilities of machine learning models to handle more diverse challenges.

The current CAPTCHA solver, designed to handle alphanumeric challenges, can be extended to address more complex CAPTCHAs involving special characters, symbols, case sensitivity, and even image-based or multi-layered CAPTCHAs like reCAPTCHA. Future improvements could enhance robustness against extreme distortions, overlapping characters, and adversarial CAPTCHAs, while transformer-based architectures like Vision Transformers could improve accuracy. Real-world deployment opportunities include developing user-friendly applications or APIs optimized for real-time performance. Additionally, exploring solutions for multimodal CAPTCHAs that require both image and text recognition, and incorporating NLP techniques for logic-based challenges, can further expand the system’s versatility to meet evolving security demands.

BIBLIOGRAPHY

1. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.

2. Jason Brownlee, *Deep Learning for Computer Vision*, Machine Learning Mastery, 2020.

3. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks." *Communications of the ACM*, 2017.

4. Relevant articles, research papers, or online resources used during the project.

    ○ TensorFlow Documentation: https://www.tensorflow.org/ ○ PyTorch Documentation: https://pytorch.org/ ○ CAPTCHA research papers: https://eprints.whiterose.ac.uk/151526/1/ccs18

APPENDIX A: TOOLS AND TECHNOLOGY

Tools

1. **Google Colab**: For training and testing the deep learning models in a cloud-based environment with GPU support.
2. **Python**: Primary programming language for data preprocessing, model development, and evaluation.
3. **Jupyter Notebooks**: For debugging, visualizing data, and documenting the project workflow.

Technologies

1. **TensorFlow/Keras**: Deep learning framework used for implementing and training the Convolutional Neural Network (CNN).
2. **NumPy and Pandas**: Libraries for data manipulation, preprocessing, and analysis.
3. **Matplotlib and Seaborn**: Tools for visualizing training metrics, dataset distributions, and model performance.
4. **OpenCV**: For image preprocessing tasks, such as noise removal, resizing, and normalization.

Hardware Requirements

- GPU-enabled system for training deep learning models efficiently.
- Storage space for datasets and model checkpoints.