School of Computer Science
Faculty of Science and Engineeing
University Of Nottingham

UG FINAL YEAR DISSERTATION REPORT

- Enhancement of Handwritten and Printed Character Recognition and Prediction using mainly Convolutional Neural Networks -

Student's name : Liqi Tang

Student Number : 20259468

Supervisor Name : Doreen Sim Ying Ying

Year : 2024

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF BACHELOR OF SCIENCE IN COMPUTER SCIENCE (HONS) THE UNIVERSITY OF NOTTINGHAM

- Enhancement of Handwritten and Printed Character Recognition and Prediction using mainly Convolutional Neural Networks -

Submitted in April 2024, in partial fulfillment of the conditions of the award of the degrees B.Sc.

- Liqi Tang-
School of Computer Science
Faculty of Science and Engineering
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature Liqi Tong

Date 02 / 05 / 2024

Acknowledgment

First and foremost, I extend my deepest gratitude to my project supervisor, Dr. Doreen, for her invaluable guidance, patience, and expert advice throughout the course of this project. Her mentorship was instrumental in shaping both the direction and success of this work.
I would like to express my appreciation to my family and friends for their unwavering support, encouragement, and understanding throughout my academic journey. Their belief in my abilities has been a source of motivation and strength.
This project would not have been possible without the collective support and encouragement of everyone mentioned above. I am immensely thankful for their contributions and for making this journey a memorable and enriching experience.

Abstract

This project focuses on enhancing an Intelligent Character Recognition (ICR) system through machine learning techniques for improved recognition of handwritten and printed characters. Beginning with dataset collection and preprocessing, including noise reduction and contrast enhancement, feature extraction methods such as Convolutional Neural Networks (CNN) are employed. Various machine learning models including Recurrent Neural Networks (RNN) like Long Short-Term Memory (LSTM) ect. are explored, with optimization techniques and ensemble methods applied to boost performance. The project aims to deliver an ICR system.

Table of Content:

CHAPTER 1: Introduction

1.1 Background Study

1.1.1 What is Intelligent Character Recognition (ICR)?

Intelligent Character Recognition (ICR) is a sophisticated form of Optical Character Recognition (OCR) technology that specialises in recognizing handwritten text. It is a handwriting recognition system that enables computers to learn and adapt to various fonts and handwriting styles during the processing phase, thereby improving accuracy and recognition capabilities independently [1]. Shortly, Intelligent Character Recognition (also known as ICR) is software that can identify fonts and styles of handwriting [2].

ICR can be viewed as a fusion of machine learning techniques and artificial intelligence that aims to simplify certain aspects of our daily routines. As an illustration, ICR technology enables the conversion of paper-based information into digital formats, allowing users to store and organise unstructured data on their computers with greater ease. By harnessing the combined capabilities of ICR and OCR engines, this technology can automatically capture data from forms and documents, eliminating the need for manual data entry. With its high accuracy and reliability, ICR proves to be an efficient method for streamlining the processing of various types of documents, ultimately saving time and effort [3].

1.1.2 How Do Intelligent Character Recognition Systems Adapt and Learn New Handwriting Patterns?

The majority of ICR software incorporates self-learning capabilities through the utilisation of neural networks. These neural networks possess the ability to automatically update the recognition database, enabling the system to identify and learn new handwriting patterns as they are encountered. Whenever the ICR system comes across a handwritten note pattern that is not currently present in its neural network, it will automatically expand its recognition database to include this new pattern. As each additional data set is processed and assimilated, the neural network's knowledge base grows, leading to an increase in the accuracy rate for recognizing handwritten notes. However, it is important to note that due to the inherent variability and lack of structure in some handwriting styles, the accuracy of the system may not be optimal in certain cases. Nonetheless, when presented with structured handwriting, ICR software can achieve an accuracy rate exceeding 97 percent. To maintain a high level of recognition accuracy, it is common practice to employ multiple reading engines within the ICR system, each contributing its selective vote to determine the final recognition result [1][4]. Across various domains, the system intelligently chooses the appropriate recognition engine based on the type of characters being processed. For instance, when dealing with numerical fields, the engine specialised in digit recognition takes precedence. Conversely, when handling text fields, the engine adept at interpreting handwritten letters is given higher priority for selection.

1.2 Problem Statement

The recognition of handwritten and printed characters is a challenging task due to the inherent variability in writing styles, font types, and language complexities. Traditional optical character recognition (OCR) systems often struggle to achieve high accuracy rates, especially when dealing with multiple languages or scripts simultaneously. This problem is compounded when recognizing characters from languages with intricate writing systems, such as Chinese and Korean, or when attempting to recognize handwritten text, which can exhibit significant variations even among characters written by the same person.

1.3 Aim and Objectives

1.3.1 Aim

The goal of this project is to develop a robust and accurate character recognition system capable of handling handwritten and printed text in Chinese, English, Korean and Japanese. The system leverages advanced machine learning techniques, such as deep learning and neural networks, to learn and recognize patterns from large datasets of labelled character images. The system will be able to adapt to different writing styles, font types, and language-specific nuances, while maintaining high accuracy rates across all three languages.

Armed with a comprehensive dataset that mirrors the vast spectrum of handwriting nuances and print conditions, this endeavour is positioned to leverage advanced machine learning algorithms, aiming to significantly enhance character recognition's accuracy, speed, and reliability. The project's ambition is not just to navigate the complexities of OCR but to redefine its capabilities, offering a sophisticated solution to the growing demand for efficient multilingual character recognition. This venture not only signifies a substantial leap forward in the digital processing of text across languages but also marks a pivotal showcase of the transformative potential within advanced computational technologies, setting a new standard for OCR systems.

1.3.2 Objectives

This character recognition project entails a comprehensive workflow aimed at recognizing characters from Chinese, English, and Korean languages. It begins with collecting and curating a diverse dataset of handwritten character images, covering various styles, fonts, and resolutions. The dataset undergoes rigorous preprocessing and normalisation for consistency. Machine learning models like CNNs and RNNs are then employed to recognize characters, utilising their feature extraction capabilities and temporal dependency capture.

Following rigorous training and evaluation, models are optimised for real-time deployment, ensuring efficient character recognition across platforms. Techniques for language detection and character segmentation enhance recognition of mixed-language text, while the development of a user-friendly interface facilitates integration into various applications,

from document digitization to language learning software, maximising accessibility and utility.

1.4 Scope and Limitations

1.4.1 Scope:

The scope of this project encompasses the development of a character recognition system capable of accurately recognizing handwritten and printed text in Chinese, English, and Korean languages. The system will leverage machine learning techniques, particularly deep learning models, to learn and recognize patterns from a dataset of labelled character images.

The project will involve the following key components:

1. Data Collection: A comprehensive dataset of handwritten character images will be collected, covering a diverse range of writing styles, font types, and image resolutions for Chinese, English, and Korean languages.
2. Data Preprocessing and Normalisation: The collected character image data will undergo preprocessing and normalisation techniques to ensure consistency and compatibility with the chosen machine learning algorithms.
3. Model Development and Training: Machine learning models, such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), will be explored and implemented for the character recognition task. These models will be trained on the prepared dataset, employing techniques like data augmentation, transfer learning, and ensemble methods to improve performance and generalisation.
4. Model Evaluation and Optimization: The trained models will be evaluated on held-out test sets, and their performance will be compared against existing OCR systems or baselines using metrics like accuracy, precision, recall, and character error rate. Model optimization techniques will be applied to ensure efficient and responsive real-time inference.
5. Language Detection and Segmentation: Techniques for language detection and language-specific character segmentation will be investigated to enable seamless recognition of mixed-language text.
6. User Interface and Integration: A user-friendly interface will be developed to facilitate the integration of the character recognition system into various applications.

1.4.2 Limitations:

While the project aims to develop a robust and accurate character recognition system, certain limitations and constraints should be acknowledged:

1. Language Coverage: The project will focus specifically on Chinese (**total in 3754 characters that are normally used in our daily life**, since Chinese characters are so large that my computer's cpu cannot handle training all of them), English, and Korean (**29 array elements** in Korean) languages. Recognition of characters from other

languages or scripts is beyond the scope of this project. For Japanese languages, I did not succeed in finding the suitable datasets.

2. Dataset Representation: Despite efforts to curate a diverse and representative dataset, it may not capture all possible variations in writing styles, font types, and image resolutions, potentially affecting the system's performance on unseen data.
3. Computational Resources: The training and deployment of deep learning models can be computationally intensive, potentially limiting the complexity of the models or the size of the dataset that can be effectively utilised within the project's resource constraints.
4. Real-world Conditions: The system's performance may be influenced by various real-world factors, such as image quality, lighting conditions, or noise, which may not be fully accounted for in the training dataset or preprocessing steps.
5. Mixed-language Text: While techniques for language detection and segmentation will be explored, the system's performance on mixed-language text containing multiple scripts or languages may be limited.
6. Integration Challenges: Integrating the character recognition system into existing applications or workflows may introduce additional challenges due to compatibility issues, user interface constraints, or specific requirements of the target applications.

CHAPTER 2: Literature Review

2.1 Overview

This literature review aims to provide a comprehensive understanding of the current state-of-the-art techniques and methodologies employed in handwritten and printed character recognition, with a focus on the application of machine learning models. The review will explore the challenges and limitations of existing approaches, highlight the recent advancements in deep learning-based character recognition, and identify potential areas for further research and improvement.

2.2 Traditional OCR Techniques:

Optical Character Recognition (OCR) has been an active area of research for several decades. Early approaches relied on traditional pattern recognition and feature extraction techniques. Cheriet et al. [10] provide a comprehensive overview of these techniques, including template matching, structural analysis, and statistical methods. Template matching involves comparing the input character image with a set of predefined templates and finding the closest match. Structural analysis, on the other hand, attempts to decompose the character into its constituent elements or strokes and recognize them based on their structural properties. Statistical techniques, such as Hidden Markov Models (HMMs) and Bayesian classifiers, model the character recognition problem as a statistical pattern recognition task [11]. While these traditional methods have been successful in certain applications, they

often struggle with variations in writing styles, font types, and image quality, limiting their performance and scalability.

2.3 Machine Learning for Character Recognition:

With the advent of machine learning techniques, character recognition systems have shown significant improvements in accuracy and robustness. Pal and Chaudhuri [12] surveyed the application of machine learning methods, such as support vector machines (SVMs), decision trees, and neural networks, for character recognition in Indian scripts. These techniques learn discriminative features from labelled training data, allowing them to generalise better to unseen variations in writing styles and font types. Plamondon and Srihari [13] provide a comprehensive review of machine learning approaches for online and offline handwriting recognition, which share many similarities with character recognition tasks. They highlight the importance of preprocessing techniques, feature extraction, and model selection in achieving accurate handwriting recognition.

2.3 Deep Learning for Character Recognition:

Deviating from the conventional approach of using recurrent neural networks followed by connectionist temporal classification for character refinement, the methodology proposed by Ptucha, R. et al. offers a distinct strategy by amalgamating the optional Lexicon CNN, Block Length CNN, and Symbol Prediction FCN. Their technique begins with a normalisation step that transforms input blocks into a standardised format, eliminating the need for resource-intensive recurrent symbol alignment correction. When a lexicon is available, a probabilistic character error rate mechanism is introduced to rectify incorrect word blocks. Unlike lexicon-centric approaches, their method can recognize common words as well as endless symbol blocks such as surnames, phone numbers, and acronyms, offering a more versatile and comprehensive handwriting recognition solution[6].

The task of converting handwritten documents into a digital format remains a significant challenge within the computer vision domain. The digitization of such documents holds immense potential for various applications, yet, it's a task often performed manually even in the modern era. Gopireddy V., Vadlamani R., and Maddula S. introduced a method of intelligent character recognition (ICR) leveraging the Resnet architecture to derive features from handwritten documents. The extracted shared features serve dual purposes - for localization and prediction. To address the scaling challenges inherent in localization, the feature pyramidal networks (FPN) were employed. Subsequently, word predictions were refined utilising a dictionary based on the Levenshtein Distance (LD) metric. Their approach exhibited promising results across three datasets, achieving an average character error rate (CER) of 12.06% and a mean Average Precision (mAP) of 0.94 [7].

C. Wigington and colleagues developed the Start, Follow, Read (SFR) model for handwriting recognition. It uses a Region Proposal Network to locate text line start points and a novel

Line Follower (LF) network to trace and preprocess curved text lines into dewarped images for recognition by a CNN-LSTM network. Building on the prior SOL method, the LF network can segment and normalise handwritten lines for input to the Handwritten Text Recognition (HWR) network. Notably, after initial pre-training, their training framework can jointly train the networks on document images using only line-level transcriptions, bypassing the need for human annotation of segmentation or spatial information which is often lacking[8].

Yuan Zhuang et al introduces a CNN-based model tailored for the recognition of handwritten Chinese characters, which incorporates median filtering for input image preprocessing to enhance image quality by smoothing and noise reduction. The model's stability and performance were evaluated through two distinct tests. The comprehensive testing revealed that the model achieved a recognition accuracy rate of approximately 90.91% following 5000 training iterations, with a final mean square error reduction to 0.0079. Additionally, the system demonstrated effective real-time performance. The incorporation of median filtering contributed to a 1.8% increase in accuracy rate over models lacking this preprocessing step, which reported a 90.48% accuracy rate. Despite the model's success, challenges such as extended training durations and the requirement for a larger dataset were identified, with the training for 5000 epochs taking 10,948 seconds. These findings underscore the potential of CNN-based models enhanced with median filtering for improving handwritten Chinese character recognition, while also highlighting areas for future optimization and research [14].

Chapter 3: RESEARCH METHODOLOGY

3.1 Overview

The research methodology will consist of several key phases, each designed to address specific aspects of the character recognition problem and contribute to the overall success of the project.

3.2 Data Collection

To develop an accurate and robust character recognition system, a diverse and comprehensive dataset will be curated, encompassing handwritten and printed characters from multiple languages, including Chinese, English, Korean, and Japanese. The dataset will be obtained from various sources, ensuring representation of different handwriting styles, fonts, and character contexts. The data collection process will prioritise diversity to enhance the model's ability to generalise across a wide range of scenarios.

Data preprocessing will be a crucial step in preparing the dataset for model training. This will involve resizing and normalising the images to ensure consistency, as well as employing data augmentation techniques to artificially increase the size and variability of the dataset. Data

cleaning and validation processes will also be implemented to ensure the quality and integrity of the training data.

3.3 Model Development

3.3.1 Dataset Preparation and Preprocessing

The collected dataset will be organised in a specific directory structure, where each folder represents a character class, and the corresponding images of that character are stored within. This structure will facilitate efficient data loading and character label assignment during model training.

3.3.2 Label Encoding

LabelEncoder from the sklearn.preprocessing module is a method utilised in machine learning and data analysis to transform categorical variables into numerical values. This technique proves beneficial when dealing with algorithms that necessitate numerical input, since the majority of machine learning models solely accept numerical data [15].

3.3.3 Model Architecture

A convolutional layer is the core component of CNNs for feature extraction. It does this through convolution, a linear operation that slides a small kernel across the input tensor. At each position, an element-wise product is computed between the kernel and input, and the results are summed into a feature map. Multiple kernels generate many feature maps capturing different input characteristics. The kernels act as feature extractors, with their size and number being critical hyperparameters defining the convolution operation [22].

A pooling layer performs a common downsampling operation, reducing the dimensionality of feature maps. This introduces translation invariance to small shifts and distortions, while decreasing the number of trainable parameters in subsequent layers. Pooling layers themselves do not have trainable parameters, but utilise hyperparameters like filter size, stride, and padding, similar to convolution operations. Max pooling, the most prevalent pooling operation, selects patches from input feature maps, retaining only the maximum value from each patch and discarding the rest. Typically, a 2x2 filter with stride 2 is used, effectively downscaling the spatial dimensions of feature maps by half. While height and width are reduced, the depth dimension remains unchanged [22].

The flatten layer plays a vital role in convolutional neural networks by bridging CNNs with ANNs, enabling the neural network models to comprehend complex patterns and generate predictions. Its primary function is to transform the feature map received from the max-pooling layer into a format compatible with dense layers. A feature map represents a multi-dimensional array containing pixel values, whereas dense layers necessitate a one-dimensional array as input for computation [23].

The __init__ function is essential for object-oriented programming in Python. It is a specific method that is automatically invoked when an object is generated from a class. Python classes employ the __init__ function, which is commonly known as the constructor. It is called automatically when an object is formed from a class and sets up the object's attributes. The term "__init__" is a Python convention that means "initialise." The syntax for specifying the __init__ method is easy. It is a special method that accepts at least one parameter, often "self," which refers to the class instance. The parameters of the __init__ method are used to accept arguments when constructing an object. When the object is instantiated, these parameters are supplied to the __init__ function [27].

The output layer of each model will be determined by the number of unique labels or character classes present in the respective recognition task. This output layer will facilitate the generation of predictions during the inference phase.

3.3.4 Model Compilation

Compiling the model in Keras involves setting up the parameters for the learning process. This includes specifying the optimizer, loss function, and metrics to be utilised. The optimizer is responsible for adjusting the network's weights to minimise the loss function, which represents what the model aims to minimise. Metrics are employed to evaluate the performance of the model [16].

The models will be compiled using the “Adam optimizer” which stands for Adaptive Moment Estimation, is an optimisation approach that builds on the strengths of two popular techniques: AdaGrad and RMSProp. Adam, like its predecessors, is an adjustable learning rate algorithm. This implies that it modifies the learning rate for each individual parameter in a model rather than employing a single global learning rate. Adam utilises several key components to optimise its performance, particularly in tasks like Intelligent Character Recognition (ICR). Firstly, it maintains an exponentially decaying average of past gradients, akin to momentum tracking in SGD, which helps in smoothing out noisy gradients and navigating intricate landscapes typical in deep learning models utilised in ICR systems. Additionally, Adam calculates an exponentially decaying average of past squared gradients, adjusting learning rates for each parameter accordingly. This adaptivity proves valuable in ICR systems, where certain character patterns may be infrequent, necessitating larger updates, while others may be frequent, requiring smaller updates. Furthermore, Adam incorporates a bias correction step to mitigate early biases in computed averages, facilitating convergence during initial training iterations. These components collectively make Adam well-suited for ICR systems, which often involve intricate deep learning models, vast datasets of character images, and the necessity to effectively handle both infrequent and frequent character patterns. Momentum tracking and bias correction further aid in navigating complex landscapes and achieving convergence in deep neural networks utilised for character recognition tasks [24].

The loss function employed will be “sparse_categorical_crossentropy”, which is well-suited for multi-class classification problems like character recognition. Sparse Categorical Cross Entropy serves as a frequently employed loss function in machine learning algorithms for training classification models. It extends the functionality of the Cross Entropy loss function, which is typically utilised for binary classification tasks [25]. Cross-validation techniques will be implemented to assess the model's robustness and generalisation capabilities on different subsets of the data. This will provide a more comprehensive understanding of the model's performance and help identify potential areas for improvement.

The metric used for evaluating the model's performance during training will be accuracy. Accuracy is an evaluation metric that measures the overall accuracy of a model's predictions [26].

3.3.5 Model Training

Model training involves providing processed data to a parameterized machine learning algorithm, resulting in a model with optimised trainable parameters aimed at minimising an objective function [17].

To enhance the model's generalisation capabilities, data augmentation techniques will be applied using the ImageDataGenerator from the Keras library. This will artificially increase the size and diversity of the training data by applying transformations such as rotation, scaling, and flipping to the input images.

The learning rate, a crucial hyperparameter that determines the step size during model optimization, will be adjusted dynamically using a learning rate scheduler. This technique helps the model converge more effectively and potentially achieve better performance.

The training process will continue for a specified number of epochs, with early stopping mechanisms in place to prevent overfitting. Early stopping allows the training to terminate when the model's performance on a validation set stops improving, ensuring that the model does not memorise the training data and retains its ability to generalise.

3.3.6 Hyperparameter Tuning

In machine learning model training, each dataset and model necessitates specific hyperparameters, which act as variables influencing the model's behaviour. Hyperparameter tuning involves conducting multiple experiments to determine the optimal hyperparameter values by sequentially training the model with different configurations. This process, crucial for enhancing model performance, can be executed manually or automated using various methods [18].

To further refine the model's effectiveness, hyperparameter tuning will be performed using Keras Tuner with RandomSearch. This will involve systematically adjusting hyperparameters like convolutional layer count, filter and kernel sizes, dense units, and dropout rates. Strategies such as early stopping, reduced learning rate on plateau, and model checkpointing will be employed to select the most effective hyperparameter configurations.

3.4 Evaluation and Validation

In Keras, the model.evaluate() function calculates the loss values for input data and, if specified during model compilation, computes any additional metrics. This function is commonly employed post-training to evaluate the performance of the model [16].

The performance of the character recognition models will be evaluated using appropriate metrics, including accuracy, precision, recall, F1-score, and confusion matrices. For multi-language recognition tasks, language-specific metrics may also be considered to assess the model's performance across different languages.

A loss function, or cost function, measures the discrepancy between a network's predicted outputs from forward propagation and the provided ground truth labels. For multiclass classification tasks, cross-entropy loss is commonly used, while mean squared error loss is often employed for regression tasks involving continuous value outputs. The choice of loss function is considered a hyperparameter and should align with the nature of the task being addressed [22].

Rigorous testing will be conducted on a separate test dataset to ensure the model's real-world applicability and generalisation. The test dataset will be carefully curated to represent a diverse range of scenarios and challenges that the character recognition system may encounter in practical applications.

3.5 Continuous Learning and Adaptation

To continuously improve the character recognition system, a feedback loop will be established. User feedback will be collected and analysed to identify specific challenges and edge cases encountered during real-world usage. This feedback will be used to refine and retrain the models, addressing the identified issues and improving the overall performance of the system.

The continuous learning and adaptation process will involve iteratively updating the dataset with new samples and scenarios, retraining the models, and deploying the improved versions of the character recognition system. This iterative approach will ensure that the system remains up-to-date and capable of adapting to evolving requirements and challenges in the field of character recognition.

Chapter4: WEBSITE DESIGN

4.1 Overview

The website is designed to provide a user-friendly interface for character recognition across multiple languages, including Chinese, English, and Korean. The primary functionality of the website revolves around uploading character images and leveraging the trained machine learning models to accurately recognize and identify the characters.

4.2 User Interface Design

The user interface design plays a crucial role in providing an intuitive and seamless experience for users interacting with the character recognition website. The design focuses on creating a clean, modern, and user-friendly interface that prioritises ease of use and accessibility.

4.2.1 Layout and Structure

The website's layout follows a straightforward and logical structure, ensuring that users can easily navigate and access the desired functionality. The main page is divided into three distinct sections, each dedicated to a specific language: Chinese Character Recognition, English Character Recognition, and Korean Character Recognition.

Within each section, the layout is consistent, featuring a header that clearly displays the language name and a brief description prompting the user to upload a character image. Underneath the header, a prominent drag-and-drop area is showcased, allowing users to intuitively upload their image files. Additionally, a "Browse files" button is provided for users who prefer traditional file selection methods.

The layout is designed to be clean and uncluttered, with ample white space and proper spacing between elements, creating a visually appealing and organised presentation. This minimalistic approach ensures that the core functionality of character recognition remains the primary focus, without any unnecessary distractions.

4.2.2 Visual Elements

The visual elements of the user interface are carefully selected to create a cohesive and aesthetically pleasing design. Typographic choices are made with readability and legibility in mind, ensuring that text elements, such as headings and descriptions, are easily readable and accessible to users with varying visual abilities.

Interactive elements, such as the drag-and-drop areas and buttons, are designed to be visually appealing and intuitive, providing clear visual cues and feedback to users during their interactions with the website.

4.2.3 Responsive Design

The layout and visual elements are optimised to adapt and scale appropriately, maintaining a consistent and usable experience regardless of the device or screen size. On smaller screens, the layout may adjust to a single-column view, ensuring that the content remains legible and the interactive elements are easily accessible. By incorporating responsive design principles, the website ensures that users can access the character recognition functionality seamlessly, regardless of their device or screen size, providing a consistent and enjoyable user experience.

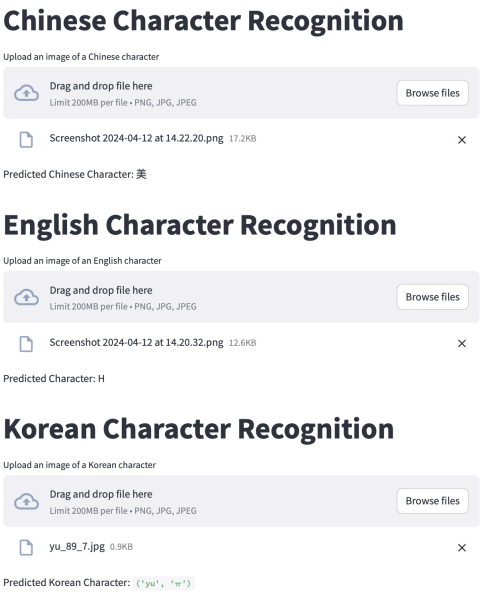


Figure 1: User Interface Design

4.3 Front-end Development

The front-end development of the character recognition website plays a crucial role in providing an intuitive and seamless user experience.

4.3.1 User Interaction and Validation

Ensuring a smooth and intuitive user experience is a key priority in front-end development. This involves implementing various interactive features and validations to guide users through the character recognition process.

The drag-and-drop functionality for file uploads is implemented using the Streamlit “st.file_uploader” function. This feature allows users to intuitively drag and drop their image files into the designated upload area, providing a seamless and user-friendly experience.

Client-side form validations are implemented to ensure that users upload compatible file formats (e.g., PNG, JPG, JPEG) and adhere to the specified file size limit. These validations

provide immediate feedback to users, preventing unnecessary uploads and improving the overall user experience.

4.3.2 File Upload and Handling

The front-end development also involves handling the file upload process, ensuring seamless communication with the back-end server for character recognition.

When a user selects or drops a file into the upload area, the front-end code captures the file data and prepares it for submission to the server.

Upon receiving a response from the server, the front-end code handles the display of the recognized character(s). This may involve updating the user interface with the recognized text or providing visual feedback indicating the successful recognition process.

Throughout the file upload and handling process, error handling mechanisms are implemented to handle any potential issues, such as network failures or server errors, and provide appropriate feedback to the user.

4.4 Back-end Development

The back-end development of the character recognition website plays a crucial role in integrating the trained machine learning models and facilitating the core functionality of character recognition.

4.4.1 Model Integration

One of the key components of the back-end development is the integration of the trained machine learning models for character recognition. These models, developed using deep learning frameworks like TensorFlow, are seamlessly integrated into the server-side application.

The integration process involves loading the pre-trained models into memory, ensuring efficient access and inference during runtime. This may involve techniques such as model serialisation, where the trained model weights and architectures are stored in a format that can be easily loaded and utilised by the server-side application.

Furthermore, the back-end includes necessary utilities and helper functions to preprocess the uploaded image data, ensuring that it is compatible with the input requirements of the machine learning models. This may involve tasks such as resizing, normalisation, and data augmentation.

4.4.2 Character Recognition Processing

The core functionality of the character recognition website revolves around processing the uploaded images and leveraging the integrated machine learning models to identify the characters present.

Upon receiving an image upload request from the user, the back-end application handles the file processing and validation. This includes checking the file format, size, and integrity to ensure compatibility and security. Once the image data is validated and preprocessed, it is passed through the integrated machine learning models for character recognition. The models analyse the image and generate predictions, identifying the characters present.

The back-end then processes these predictions, potentially applying post-processing techniques or additional logic as required. The recognized characters are then formatted and prepared for presentation to the user.

4.5 Use Case Diagram

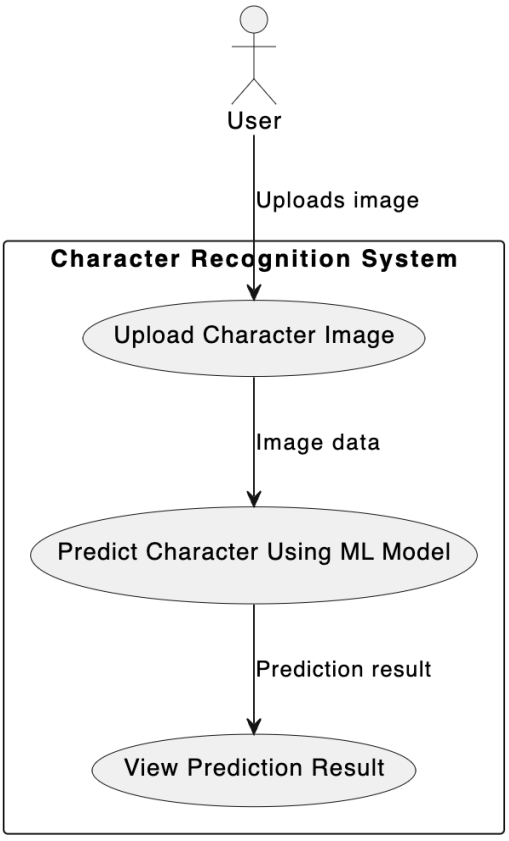


Figure 2: use case diagram

4.6 Sequence Diagram

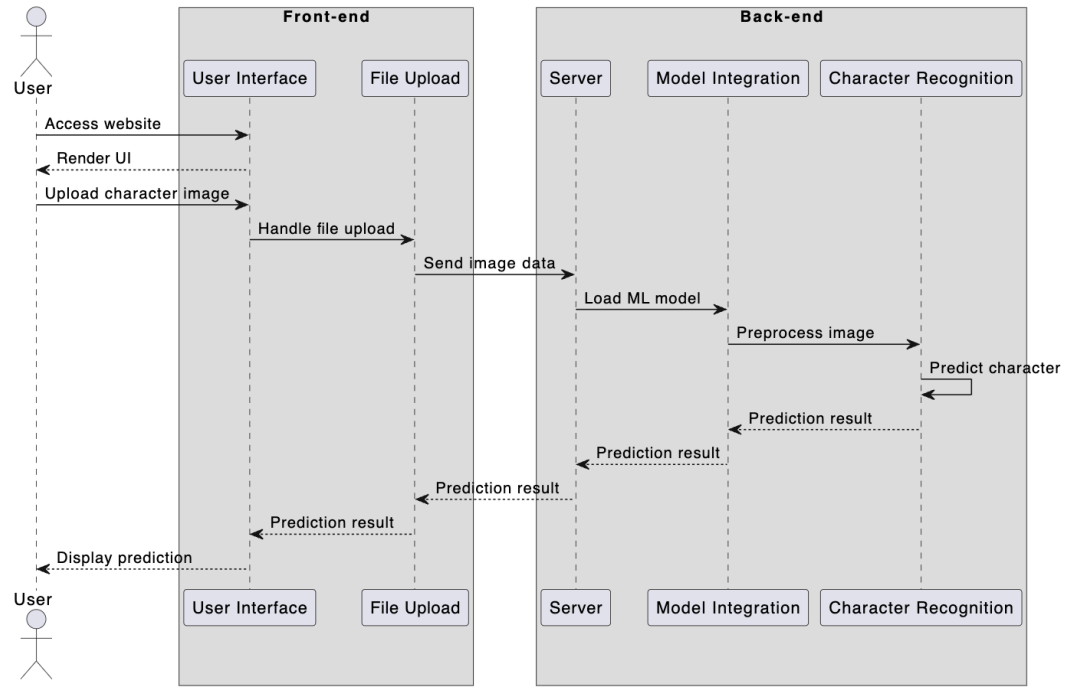


Figure 3: sequence diagram

Chapter 5: RESULTS AND DISCUSSION

5.1 Overview

The implementation of the character recognition system involves the development and deployment of machine learning models for three different languages: Chinese, English, and Korean. Each language poses its own unique challenges and considerations, requiring tailored approaches to data preprocessing, model architecture, and training strategies.

5.2 Chinese Character

5.2.1 Dataset Preparation

The initial dataset used for Chinese character recognition was the CASIA-HWDB dataset from Kaggle, which is approximately 6GB in size and includes more than 7,000 Chinese characters. However, due to the large size of the dataset, causing issues with memory limitations, the dataset was modified to include only the Train file for both training and testing steps. Additionally, punctuation marks, numbers, and English characters were removed from the original CASIA-HWDB dataset.

While working with the CASIA-HWDB dataset, you encountered challenges in achieving good accuracy using both the CNN model and the hyper model, with the accuracy being only

around 0.01%. This could be due to various factors, including the complexity of the dataset, the model architecture, or the training process.

To overcome this limitation and the constraints imposed by my CPU memory, I decided to explore an alternative dataset. I found a new dataset online that contains 3754 frequently used Chinese characters, which are commonly encountered in daily life.

By switching to this more focused and practical dataset, which aims to improve the accuracy of my character recognition system while also addressing the memory constraints of my computational resources. The new dataset, containing a curated selection of commonly used Chinese characters, is expected to be more manageable in terms of size and complexity, potentially leading to better training performance and higher accuracy.

Additionally, the inclusion of frequently used characters aligns with the real-world application of the character recognition system, ensuring that it can effectively handle the characters encountered in everyday situations.

This strategic decision to change the dataset demonstrates your ability to adapt and explore alternative solutions when faced with challenges. By prioritising practical considerations and targeting a more focused set of characters, I have increased the likelihood of achieving better results and developing a more robust and reliable character recognition system tailored to real-world needs.

Data loading, preprocessing, label encoding, and data augmentation processes are critical steps in preparing datasets for training and evaluating machine learning models, particularly in image classification tasks. The dataset is organised in a directory structure where each folder represents a character class, and the images within each folder are preprocessed to grayscale, resized to 64x64 pixels, and normalised. Labels are extracted from folder names, and label encoding is applied to represent categorical character labels numerically.

Data augmentation techniques, facilitated by the ImageDataGenerator class, are employed to generate augmented batches of images and labels for training and validation. These processes are essential for enhancing the model's performance, improving generalisation, and mitigating overfitting by enriching the dataset with diverse variations of existing data. For a comprehensive understanding of the data preprocessing and analysis methodologies employed, please consult Appendix I.

5.2.2 Basic CNN Model for CASIA-HWDB dataset

The basic CNN model for Chinese character recognition is implemented using TensorFlow and Keras. The model architecture is constructed using the Sequential API, which allows for stacking multiple layers in a sequential manner. To gain insight into the intricate details

pertaining to the Model Architecture, Model Compilation, and Model Training processes, kindly refer to the comprehensive information provided in Appendix II.

The final model is evaluated on the test set to determine its accuracy and loss on unseen data.

```
# Step 6: Evaluate the model
test_loss, test_acc = model.evaluate(X_val, y_val)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_acc * 100:.2f}%')
```

```
4638/4638 [==============================] - 67s 14ms/step - loss: 8.8095 - accuracy: 1.0781e-04
Test Loss: 8.809479713439941
Test Accuracy: 0.01%
```

Figure 4. Loss and Accuracy Value

Loss and accuracy curves are plotted to visualise the model's learning behaviour during training.

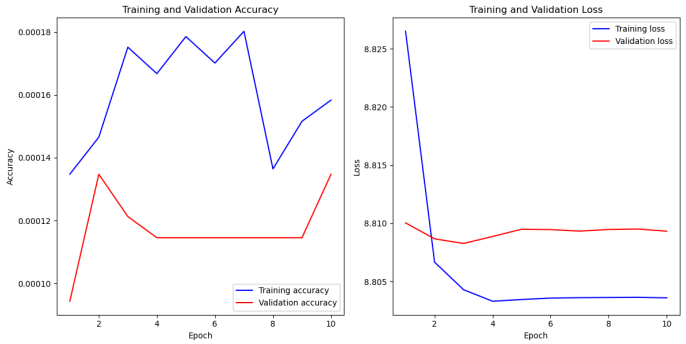


Figure 5. Loss and accuracy curves

5.2.3 CNN model with more focused and practical dataset

"CustomCNN" class encapsulates a convolutional neural network architecture tailored for character recognition tasks. The model architecture comprises feature extraction layers, including convolutional and max-pooling layers, followed by integration and classification layers. The convolutional layers are designed to extract essential features such as edges and patterns from input character images, while the max-pooling layers reduce spatial dimensions and enhance computational efficiency. The flatten layer consolidates the extracted features into a one-dimensional vector, preparing them for classification. Finally, the dense layer performs the classification task, outputting predicted probabilities for each character class using the softmax activation function. By defining the forward pass through the call method, the model seamlessly processes input data through the defined layers, ultimately producing accurate predictions.

Overall, the CustomCNN architecture offers a structured and effective framework for character recognition tasks, leveraging the power of CNNs to extract meaningful features and classify characters with high accuracy. To gain insight into the intricate details pertaining to the Model Architecture, Model Compilation, and Model Training processes, kindly refer to the comprehensive information provided in Appendix III.

The performance of the basic CNN model is evaluated on the test set, and the following visualisations are provided:

The model.evaluate(test_generator) method evaluates the trained model's performance on the test data generated by test_generator. The evaluation assigns test_loss and test_acc variables, with test_acc being 63.93% - the model's accuracy on the test set. These metrics assess the model's generalisation ability, aiding decisions like model selection and deployment readiness.

```
5611/5611 [==============================] - 52s 9ms/step - loss: 1.7304 - accuracy: 0.6393
Test Loss: 1.7303780317306519
Test Accuracy: 63.93%
```

Figure 6. Loss and accuracy value

The function plot_loss_and_accuracy_curves is being called to print the test_loss and test_acc values, to see the performance of the model on the test dataset.

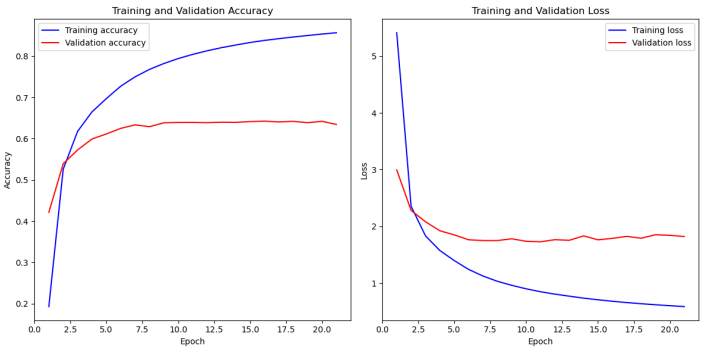
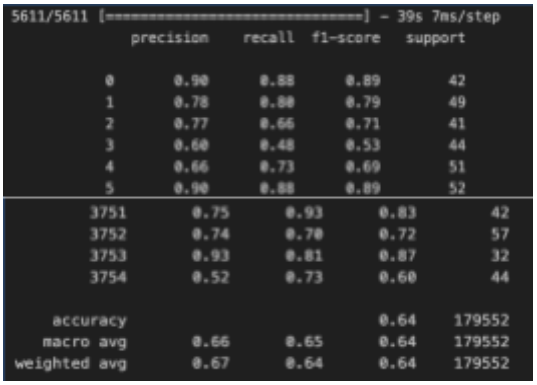


Figure 7. Loss and accuracy curve

Predict the validation set y_pred by using X_val set, get the predicted labels y_pred_labels and select the indices of the maximum values along axis 1 (across rows) by using y_pred. Compute the confusion matrix using the true labels (y_val_encoded) and the predicted labels (y_pred_labels), which is used to describe the performance of a classification model, showing the actual versus predicted classifications and highlighting where the model is getting things right or wrong. Finally, print the classification_report(y_val_encoded, y_pred_labels) which compares the true labels (y_val_encoded) with the predicted labels (y_pred_labels) and calculates various metrics to evaluate the model's performance, such as precision, recall, F1-score, and support for each class.



```
5611/5611 [==============================] - 39s 7ms/step
              precision    recall  f1-score   support

           0       0.90      0.88      0.89        42
           1       0.78      0.80      0.79        49
           2       0.77      0.66      0.71        41
           3       0.60      0.48      0.53        44
           4       0.66      0.73      0.69        51
           5       0.90      0.88      0.89        52

        3751       0.75      0.93      0.83        42
        3752       0.74      0.70      0.72        57
        3753       0.93      0.81      0.87        32
        3754       0.52      0.73      0.60        44

    accuracy                           0.64    179552
   macro avg       0.66      0.65      0.64    179552
weighted avg       0.67      0.64      0.64    179552
```

Figure 8. Part of classification report of confusion matrix

5.2.4 CNN model with Median filter to images

While training the previous CNN model, I noticed a substantial variation in training and validation metrics such as loss and accuracy, which might be due to overfitting or underfitting.

Overfitting arises when a model excels on training data but falters on test data, having learned the noise inherent to the training set instead of generalising well. It stems from low bias and high variance. Conversely, underfitting happens when a model fails to capture the patterns in training data adequately, leading to poor performance across datasets. An underfit model exhibits high bias and low variance, unable to model complexities sufficiently [28]. Due to the previous loss and accuracy curves shown in the final model, I can readily determine that it is underfitting.

Underfitting in machine learning models can occur for several reasons, each impacting the model's ability to generalise well from the training data to unseen data. One common cause is uncleaned training data that includes noise or garbage values, which can confuse the model during the learning process. Additionally, underfitting can arise from models that inherently have a high bias, meaning they are too simplistic to capture the underlying patterns in the data. Sometimes, the sheer volume of the training data may also be insufficient, not providing enough variability or examples for the model to learn effectively. Lastly, the architectural simplicity of a model—where it doesn't have the necessary depth or complexity—can lead to underfitting [28].

To tackle underfitting, several strategies can be employed. Enhancing the dataset with more features can provide the model with additional information that might be crucial for making accurate predictions. Increasing the model's complexity allows for a richer learning capacity, enabling it to understand more complex patterns. Cleaning the dataset to reduce noise will help the model focus on the true signals in the data. Furthermore, extending the duration of training gives the model more opportunity to learn from the data, potentially uncovering patterns it missed in shorter training runs. By addressing these aspects, one can significantly improve a model's performance and its ability to generalise [28].

So in order to handle the underfitting of CNN model, I would like to reduce the image noise by applying the median filter here in this model to improve the accuracy.

The median filter excels at reducing random noise, especially long-tailed or periodic noise patterns. It slides a window across the image, calculating the median value within each window to populate the corresponding output pixel. For Laplacian noise, the median provides the best location estimate. The filter preserves reasonably uniform areas while adeptly handling long-tailed noise. At edges, the window's dominant side causes an abrupt output transition between intensity values [29].

5.2.4.1 Apply Median Filter to Images

“apply_median_filter” function is using a built-in function called “median_filter” to apply the filter to the input image. The median_filter function takes two arguments: the image to be filtered and the size of the filter. In this case, the filter size is set to 3.

An instance of the ImageDataGenerator class is created with train_generator = ImageDataGenerator(preprocessing_function=apply_median_filter).flow(X_train, y_train_encoded), specifying apply_median_filter as the preprocessing function and using X_train and y_train_encoded as the training data inputs. Similarly, another ImageDataGenerator instance is instantiated for the validation data (X_val, y_val_encoded), applying the same preprocessing function. The flow method is called on these generators to create batches of augmented and preprocessed images from the provided training and validation data for model training and evaluation.

To begin, the saved CNN model is loaded from the desktop. Next, model.compile configures the model for training by specifying the Adam optimizer, sparse_categorical_crossentropy loss function suitable for multi-class classification problems with integer labels, and accuracy as the evaluation metric. Early stopping (early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)) and reduce learning rate (reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)) callbacks are defined to monitor validation loss and adjust learning rates accordingly during training. Finally, the model is fit on the training data, similar to the previous CNN model, utilising the defined callbacks and optimizations for potentially improved performance.

The performance of this model is evaluated on the test set, and the following visualisations are provided:

The evaluate method returns the loss value and the accuracy of the model on the test dataset which is 68.06%.

```
5611/5611 [==============================] - 68s 12ms/step - loss: 1.5360 - accuracy: 0.6806
Test Loss: 1.535983681678772
Test Accuracy: 68.06%
```

Figure 9. Loss and accuracy value

The function plot_loss_and_accuracy_curves is being called to print the test_loss and test_acc values, to see the performance of the model on the test dataset.

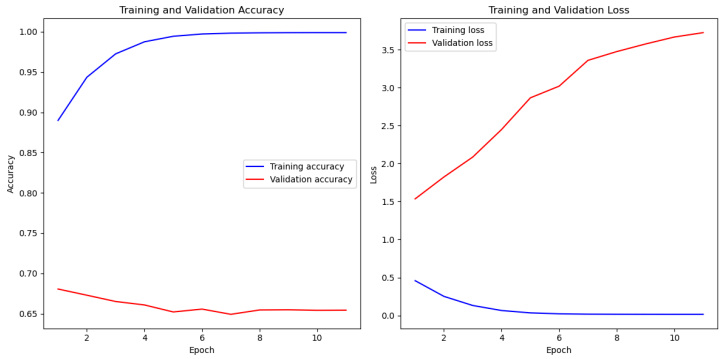


Figure 10. Loss and accuracy curve

Print the classification_report(y_val_encoded, y_pred_labels) which compares the true labels (y_val_encoded) with the predicted labels (y_pred_labels) and calculates various metrics to evaluate the model's performance.

	precision	recall	f1-score	support
0	0.89	0.93	0.91	42
1	0.79	0.84	0.81	49
2	0.61	0.66	0.64	41
3	0.74	0.39	0.51	44
4	0.59	0.84	0.69	51
5	0.85	0.88	0.87	52
6	0.94	0.69	0.80	48
7	0.84	0.69	0.76	55
8	0.86	0.81	0.83	59
9	0.72	0.57	0.63	46
10	0.78	0.68	0.73	41
11	0.83	0.63	0.72	54
12	0.69	0.54	0.60	41
13	0.70	0.73	0.72	52
14	0.70	0.59	0.64	44
15	0.75	0.45	0.56	47
16	0.70	0.84	0.76	38
17	0.88	0.84	0.86	44
18	0.45	0.63	0.52	35
19	0.87	0.64	0.74	42
20	0.86	0.59	0.70	51
21	0.69	0.64	0.67	42
22	0.73	0.84	0.78	43
...				
accuracy			0.66	179552
macro avg	0.69	0.67	0.66	179552
weighted avg	0.69	0.66	0.67	179552

Figure 11. Part of classification report of confusion matrix

5.2.5 Updated CNN model

Another strategy described before that enhances the model's complexity provides for a deeper learning capacity, allowing it to recognise more complicated patterns. This may also be used to improve the accuracy of the CNN model. For example, in this case I increase the number of CNN layers, meanwhile keep using the datasets after the median filter to increase the accuracy.

5.2.5.1 Updated CNN model 1

The `CustomCNN_2` class is a subclass of `models.Model`. Within this class, there exists an `__init__` method, serving as the constructor. It initialises when an instance of the class is

created, optionally taking the parameter `number_of_characters`, which determines the number of output units in the final dense layer of the model. The architecture includes a total of four layers: three convolutional layers followed by one fully connected layer. For comprehensive information regarding the layers and model compilation, please consult Appendix IV.

The performance of this Customic CNN model is evaluated on the test set, and the following visualisations are provided:

The evaluate method returns the loss value and the accuracy of the model on the test dataset which is up to 75.59%.

```
5611/5611 [==============================] - 64s 11ms/step - loss: 1.0284 - accuracy: 0.7559
Test Loss: 1.0283708572387695
Test Accuracy: 75.59%
```

Figure 12. Loss and accuracy value

The function plot_loss_and_accuracy_curves is being called to print the test_loss and test_acc values, to see the performance of the model on the test dataset.

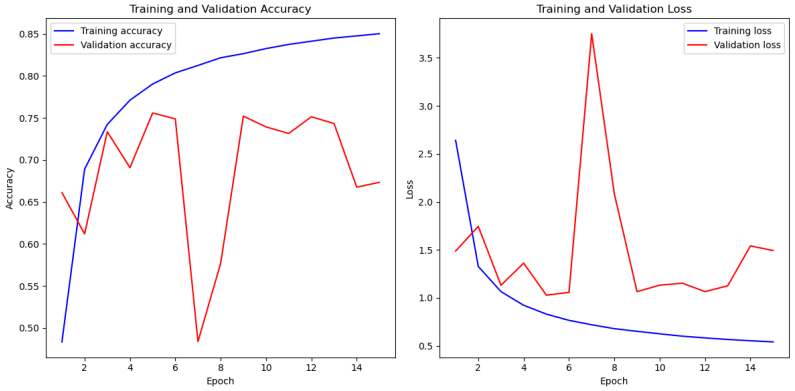


Figure 13. Loss and accuracy curve

Print the classification_report(y_val_encoded, y_pred_labels) which compares the true labels (y_val_encoded) with the predicted labels (y_pred_labels) and calculates various metrics to evaluate the model's performance.

```
5611/5611 [==============================] - 34s 6ms/step
              precision    recall  f1-score   support

           0       0.87      0.98      0.92        42
           1       0.79      0.86      0.82        49
           2       0.76      0.85      0.80        41
           3       0.77      0.55      0.64        44
           4       0.69      0.73      0.70        51
           5       0.81      0.88      0.84        52
           6       0.66      0.96      0.78        48
           7       0.84      0.78      0.81        55
           8       0.78      0.92      0.84        59
           9       0.78      0.76      0.77        46
          10       0.79      0.80      0.80        41
          11       0.91      0.78      0.84        54
          12       0.58      0.68      0.63        41
          13       0.90      0.67      0.77        52
          14       0.68      0.73      0.70        44
          15       0.77      0.64      0.70        47
          16       0.60      0.92      0.73        38
          17       0.97      0.89      0.93        44
          18       0.72      0.83      0.77        35
          19       1.00      0.71      0.83        42
          20       0.88      0.71      0.78        51
          21       0.73      0.76      0.74        42
...
    accuracy                           0.75    179552
   macro avg       0.77      0.75      0.75    179552
weighted avg       0.77      0.75      0.75    179552
```

Figure 14.Part of classification report of confusion matrix

5.2.8.2 Updated CNN model 2

Try to add one more CNN layer on the model and see whether the accuracy increases or not.

The `CustomCNN_3` class has a constructor method called __init__, which is executed when an object of the class is created. The constructor takes an optional parameter number_of_characters, which has a default value of 20. This parameter represents the number of output classes or categories that the model will predict.
An additional fourth convolutional layer has been introduced to the network. This layer conducts a convolution operation using 48 filters, each with a dimension of 3x3, and is subsequently followed by a batch normalisation process. This is designed to standardise the activations from the convolutional layer, aiding in stabilising the learning process. Following this, the fourth pooling layer performs a max pooling operation, effectively reducing the spatial dimensions of the resulting feature maps by selecting the maximum value in each pooling window. To further mitigate the risk of overfitting, a fourth dropout layer is included. This layer operates similarly to its predecessors by randomly deactivating a specified proportion of its input nodes, introducing redundancy into the network that encourages the learning of more robust features.

The performance of this updated Customic CNN model is evaluated on the test set, and the following visualisations are provided:

The evaluate method returns the loss value and the accuracy of the model on the test dataset which is 82.70%.

```
5611/5611 [==============================] - 67s 12ms/step - loss: 0.6872 - accuracy: 0.8270
Test Loss: 0.6872409582138062
Test Accuracy: 82.70%
```

Figure 15. Loss and accuracy value

The function plot_loss_and_accuracy_curves is being called to print the test_loss and test_acc values, to see the performance of the model on the test dataset.

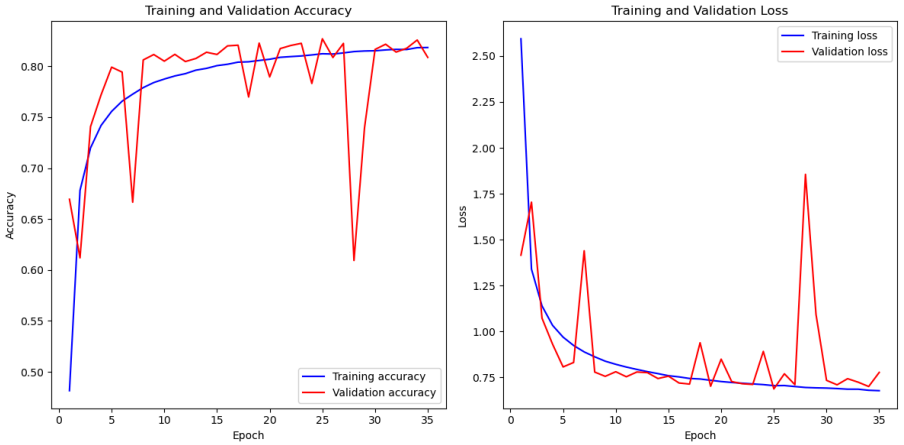


Figure 17. Loss and accuracy curve

Print the classification_report(y_val_encoded, y_pred_labels) which compares the true labels (y_val_encoded) with the predicted labels (y_pred_labels) and calculates various metrics to evaluate the model's performance.

```
5611/5611 [==============================] - 37s 7ms/step
              precision    recall  f1-score   support

           0       1.00      0.71      0.83        42
           1       0.76      0.96      0.85        49
           2       0.80      0.90      0.85        41
           3       0.73      0.73      0.73        44
           4       0.95      0.82      0.88        51
           5       0.96      0.83      0.89        52
           6       0.78      0.90      0.83        48
           7       0.88      0.89      0.88        55
           8       0.87      0.93      0.90        59
           9       0.64      0.85      0.73        46
          10       0.97      0.88      0.92        41
          11       0.80      0.87      0.83        54
          12       0.74      0.90      0.81        41
          13       0.94      0.94      0.94        52
          14       0.96      0.61      0.75        44
          15       0.78      0.83      0.80        47
          16       0.74      0.92      0.82        38
          17       0.93      0.98      0.96        44
          18       0.69      0.77      0.73        35
          19       0.89      0.93      0.91        42
          20       0.86      0.82      0.84        51
          21       0.81      0.83      0.82        42
...
    accuracy                           0.83    179552
   macro avg       0.84      0.83      0.83    179552
weighted avg       0.84      0.83      0.83    179552
```

Figure 17.Part of classification report of confusion matrix

5.2.6 Summary

In summarising the results for models trained on Chinese characters, it is evident that the implementation of a median filter on the datasets results in a notable improvement in accuracy, achieving a 4% increase to 68%. This enhancement is confirmed through both evaluative metrics and analysis via a confusion matrix. On the other hand, the addition of an extra convolutional layer to the basic CNN model, in conjunction with the median filter, led to a further increase in accuracy to 75.59%, surpassing the median filter's standalone application by 7%. Moreover, the introduction of an additional CNN layer boosted the model's accuracy significantly to 82.7%, a substantial rise from the basic model's accuracy of 63.93%. Nonetheless, the extensive size of the datasets poses a challenge for the use of

hyperparameter-tuned models due to the prolonged training durations; consequently, such models were not employed in the training of the Chinese characters model.

5.3 English Character

5.3.1 Dataset Preparation and Preprocessing

The dataset is imported from a CSV file containing images' corresponding labels, read and stored into a DataFrame 'df'. Create a 'base_image_dir' to the base directory where the images are stored. A custom function 'load_and_preprocess_image' is defined to load images, convert them to RGB format, resize them to 128x128 pixels, and normalise the pixel values. Two empty lists, images and labels, are initialised to store the preprocessed image data and corresponding labels, respectively.

Iterates over each row in the 'df' using the `iterrows()` method. This DataFrame.iterrows() function iterates across Pandas DataFrame rows using (index, Series) pairs. The iterator delivers a tuple of column names and data formatted as a series [30]. For each row, it retrieves the image path by joining the base_img_dir with the `row['image']` value , and it retrieves the label for the image from `row['label']`. Using the 'load_and_preprocess_image' function, it loads and preprocesses the image located at the image_path. The preprocessed image array (img_array) and the corresponding label are appended to the images and labels lists, respectively. After iterating over all rows, the `images` and `labels` lists are converted to NumPy arrays using `np.array(images)` and `np.array(labels)`, respectively.

The variables 'X_train', 'X_test, y_train', and 'y_test' will store the split datasets, where `X` represents the input features or independent variables (images), and `y` represents the target variable or dependent variable (labels).

The `train_test_split` function from scikit-learn is then called, passing in the `images` and `labels` arrays, along with the `test_size=0.2` argument to specify that 20% of the data should be allocated for testing. The `random_state=42` argument ensures reproducibility by setting the random seed for the split. The `train_test_split` function returns four arrays: `X_train`, `X_test`, `y_train`, and `y_test`, representing the training and testing data for the input features and target variables, respectively.

5.3.2 Label Encoding

A. An instance of the LabelEncoder class from scikit-learn is created and assigned to the variable 'label_encoder'. This object will be used to encode the labels into integers.
B. The 'fit_transform' method is called on the 'label_encoder' object, passing in the training labels y_train. This method first learns (or "fits") the unique labels present in the training data and then transforms those labels into integer values. The encoded training labels are stored in the y_train_encoded variable.

C. For the test labels y_test, only the transform method is called on the label_encoder object. This is because the label encoder has already been fitted on the training data, and also wants to use the same mapping to encode the test labels. The encoded test labels are stored in the y_test_encoded variable.

After this step, the encoded training labels (y_train_encoded) and encoded test labels (y_test_encoded) can be used for model training and evaluation, respectively.

5.3.3 CNN with GRU model

GRU (Gated Recurrent Unit) is an RNN architecture similar to LSTM, designed to model sequential data by selectively retaining or forgetting information over time. Simpler than LSTM with fewer parameters, GRU trains faster and is more computationally efficient. Instead of a memory cell state, GRU has a "candidate activation vector" updated by two gates: the reset gate, determining how much previous hidden state to forget, and the update gate, deciding how much of the candidate vector to incorporate into the current hidden state [31].

The model comprises a total of 8 layers, distributed as follows: 3 convolutional layers, 3 max pooling layers, 1 integration layer, and 1 sequential processing layer (named gru). Further information regarding the model's architecture and compilation can be found in Appendix V.

The performance of this GRU with CNN model is evaluated on the test set, and the following visualisations are provided:

Training and validation accuracy and loss are monitored and visualised using Matplotlib to assess model performance.

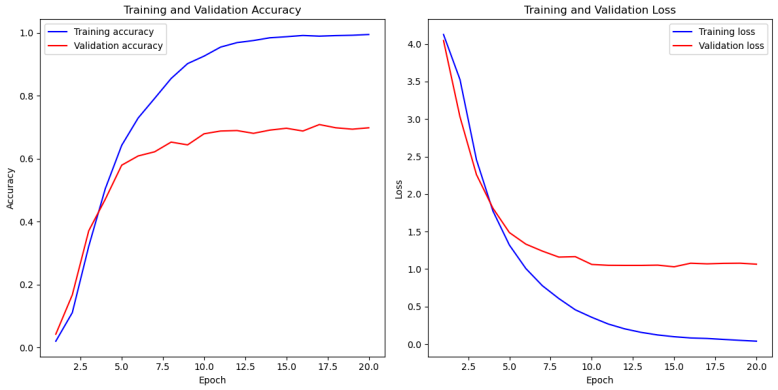
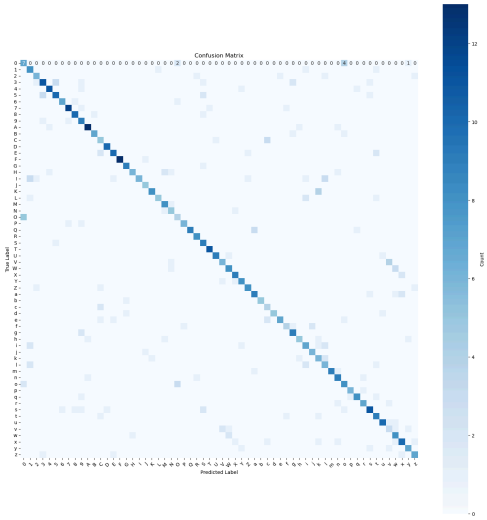


Figure 18: Loss and accuracy curves

Print the classification_report(y_val_encoded, y_pred_labels) matrices and calculates various metrics to evaluate the model's performance.



accuracy			0.71	682
macro avg	0.73	0.72	0.71	682
weighted avg	0.73	0.71	0.71	682

Figure 19: Confusion matrix

And loss value and the accuracy of the model: 69.65%

```
22/22 [==============================] - 1s 58ms/step - loss: 1.0308 - accuracy: 0.6965
Test Loss: 1.0308376550674438
Test Accuracy: 69.65%
```

Figure 20: loss value and the accuracy

5.3.4 CNN with GRU model with ImageDataGenerator

An instance of the `ImageDataGenerator` class is initialised to facilitate data augmentation. The `datagen` object is configured with several parameters: `width_shift_range`, `height_shift_range`, `shear_range`, and `zoom_range`, which respectively determine the permissible horizontal and vertical shifting, shear transformations, and zooming for the images. Specifically, these parameters are set to 0.01, 0.01, 0.1, and 0.1, allowing for horizontal and vertical shifts of up to 1% of image dimensions, shear transformations of up to 10%, and zooming in or out by up to 10%.

Subsequently, augmented versions of the training and test sets are generated using the `flow` method on the `datagen` object. The generated data is returned as a generator object, enabling its utilisation in training or evaluating machine learning models. The original training data `X_train` along with their corresponding encoded labels `y_train_encoded` are used to produce augmented training samples, which are stored in a new variable named `train_generator`. Similarly, the original test data `X_test` and their encoded labels `y_test_encoded` are utilised to generate augmented test samples, stored in a variable named `test_generator`. Each batch of the generated data contains 32 samples, as specified by the `batch_size` parameter.

For further details on the training of the model, please refer to Appendix VI.

The performance of this this model is evaluated on the test set, and the following visualisations are provided:

Training and validation accuracy and loss are monitored and visualised using Matplotlib to assess model performance.

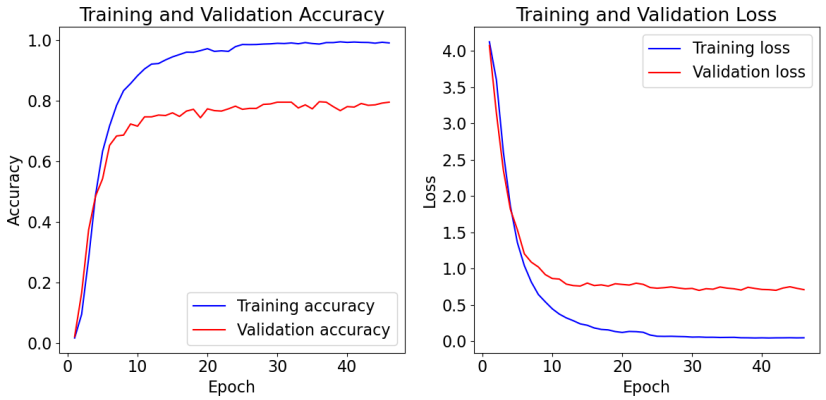
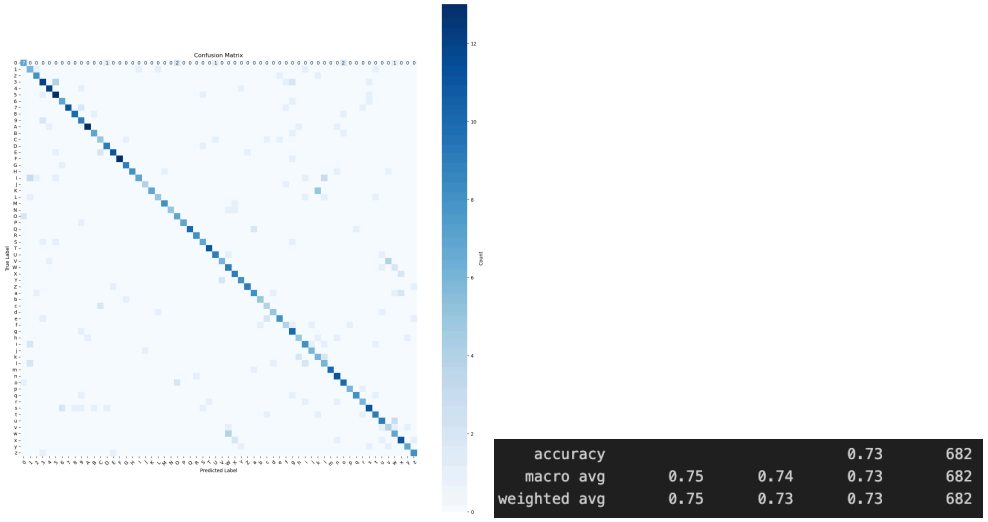


Figure 21: Loss and accuracy curves

Print the classification_report(y_val_encoded, y_pred_labels) matrices and calculates various metrics to evaluate the model's performance.



accuracy			0.73	682
macro avg	0.75	0.74	0.73	682
weighted avg	0.75	0.73	0.73	682

Figure 22: Confusion matrix

With Higher Evaluation Accuracy:71.41%

```
22/22 [==============================] - 2s 92ms/step - loss: 0.9918 - accuracy: 0.7141
Test Loss: 0.9917590618133545
Test Accuracy: 71.41%
```

Figure 23: Evaluation Accuracy

Using the image data generator, we can determine more accuracy (2% improvement) and less underfitting based on the performance of two loss and accuracy curves.

5.3.5 CNN with GRU model by using Hyperparameter Tuning

To improve the model's performance, hyperparameter tuning is performed using Keras Turner's RandomSearch. This process involves systematically searching for the optimal configuration of various hyperparameters, such as the number of convolutional layers, filter

sizes, kernel sizes, and the number of units in dense layers. After conducting 15 trials, the accuracy is improved to 75.22%.

The ICRHyperModel class is meticulously designed to create a flexible and adaptable convolutional recurrent neural network (CRNN) architecture specifically for image-based character recognition tasks. Utilising the powerful Keras functional API, this model dynamically builds a deep learning architecture that can accommodate a variable number of convolutional layers, determined by a hyperparameter. Each convolutional layer is highly configurable, allowing adjustments to the number of filters and kernel size, enabling the model to extract intricate features from input images effectively. After convolutional processing, the feature maps are reshaped to prepare them for processing by recurrent layers, where Gated Recurrent Unit (GRU) layers skillfully capture sequential patterns within the flattened feature space. The output from the recurrent layers is then passed through a fully connected layer with a softmax activation function, allowing the model to generate probabilistic predictions across multiple character classes. With its dynamic and adaptable architecture, this model serves as a powerful tool for image-based character recognition tasks, offering remarkable flexibility and performance. For a comprehensive understanding of the model's intricacies and implementation details, please refer to Appendix VII.

To initialise the hyperModel and Tuner. First, set the input shape for a model. The input shape is a 3-dimensional tuple with the dimensions (128, 128, 3). This implies that the model expects input data with a height and width of 128 pixels and three colour channels (RGB). Determines the number of classes in the classification issue. In this scenario, the model will be trained to categorise 62 different classes.

Initialises a HyperModel object called `hypermodel`. It takes two arguments: `input_shape` and `num_classes`. The HyperModel is a custom class that most likely specifies the architecture of the model to be adjusted. Creates a RandomSearch tuner object named ‘tuner’. The RandomSearch tuner is used to find the optimal hyperparameters for the model. It requires multiple arguments:

A. objective: The metric to optimise during the hyperparameter search. Set to 'val_accuracy', which suggests that the tuner will try to maximise the validation accuracy.

B. max_trials: The maximum number of hyperparameter combinations to try during the search. It is set to 15.

C. directory: The directory where the tuner will save its results and checkpoints. It is set to 'model_tuning'.

D. project_name: The name of the tuning project. In this case, it is set to 'english_tuning'.

The performance of this best hypermeter model is evaluated on the test set, and the following visualisations are provided:

Training and validation accuracy and loss are monitored and visualised using Matplotlib to assess model performance.

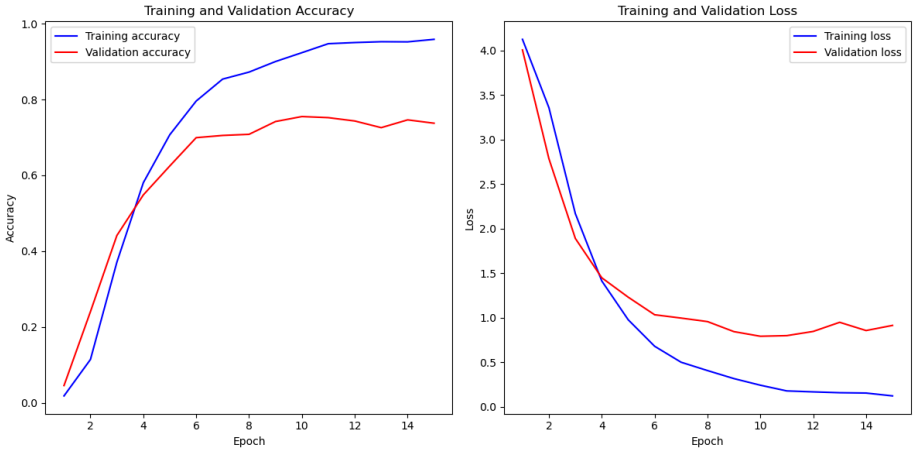
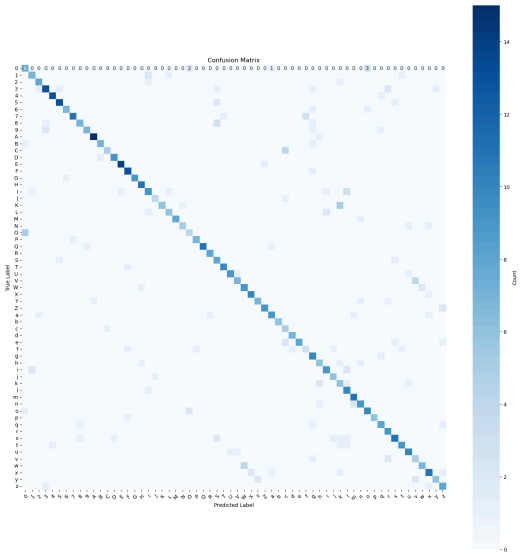


Figure 24: Loss and accuracy curves

Print the classification_report(y_val_encoded, y_pred_labels) matrices and calculates various metrics to evaluate the model's performance.



accuracy			0.76	682
macro avg	0.78	0.77	0.76	682
weighted avg	0.78	0.76	0.76	682

Figure 25: Confusion matrix

With Higher Evaluation Accuracy:75.22%

```
22/22 [==============================] - 10s 474ms/step - loss: 0.8085 - accuracy: 0.7522
Test Loss: 0.8085266351699829
Test Accuracy: 75.22%
```

Figure 26: Evaluation Accuracy

5.3.6 CNN Model with l2 Regularisation

Regularisation is a machine learning and statistical modelling strategy that prevents overfitting and improves model generalizability. When a model is overfitting, it has learnt too much from the training data and may struggle to perform well on fresh, untested data. Regularisation adds restrictions or penalties to the model throughout the training process, with the goal of controlling the model's complexity and avoiding over-reliance on certain training data characteristics or patterns. Regularisation helps to establish a balance between fitting the training data effectively and generalising it to fresh data [33].

L2 regularisation, also known as Ridge regularisation, adds the squared values of the model's coefficients to the loss function. L2 regularisation does not need the coefficients to be absolutely zero, but rather promotes them to be modest. L2 regularisation prevents overfitting by distributing the effect of a single feature across numerous features. It is favourable when the input properties are correlated [33].

L2 regularisation produces reduced but non-zero coefficients for all characteristics. It lowers the influence of particular characteristics while without causing coefficients to zero. The L2 penalty is useful when dealing with high feature correlations or when no special feature selection is required, because it enables all features to contribute to the model's predictions. The L2 regularisation term is smooth and differentiable, making optimisation using typical gradient-based techniques computationally fast [33].
First, define a function called `build_icr_model_with_regu` that is used to construct a model for image classification with regularisation.

The model begins with an Input layer defined as `input_img = Input(shape=self.input_shape, name='image_input')`, which sets the input shape based on the specified dimensions during class instantiation. Following this is a series of convolutional layers, each employing the ReLU (rectified linear unit) activation function via activation='relu' to introduce non-linearity. The padding='same' parameter ensures that the spatial dimensions of the output feature maps match those of the input, while kernel_regularizer=l2(l2_rate) applies L2 regularisation to the convolutional kernel weights, helping prevent overfitting. After each convolutional layer, a MaxPooling2D layer is added to progressively reduce the spatial dimensions of the feature maps through max pooling operations.

Following the convolutional and pooling layers, a dropout layer (Dropout) is added. Dropout randomly sets a fraction of input units to 0 during training, which helps to prevent overfitting by introducing noise and reducing interdependencies between neurons. The output of the dropout layer is then reshaped using the Reshape layer. This is done to prepare the output for the recurrent neural network (RNN) layers. The function adds a GRU (Gated Recurrent Unit) layer to the model. An RNN is a form of neural network that can process sequences such as text, voice, movies, financial data, and more. Combining CNNs and RNNs

allows us to work with pictures and word sequences in this situation. The purpose is to create captions for a given photograph [34]. In this case, it is used to process the reshaped output from the previous step. The `return_sequences=False` parameter indicates that the GRU layer should only return the output at the last time step. Another dropout layer is added after the GRU layer to further prevent overfitting.

Finally, a fully connected layer (Dense) is added to the model. This layer maps the output of the GRU layer to the number of classes specified by num_classes. The activation='softmax' parameter applies the softmax activation function, which converts the model's output into a probability distribution over the classes. The kernel_regularizer=l2(l2_rate) parameter applies L2 regularisation to the weights of the fully connected layer. The function then creates an instance of the Model class from Keras, specifying the input and output layers. Finally, the function returns the constructed model. For a deeper understanding of the model compilation and training procedures, please consult Appendix VIII for detailed insights and instructions.

The performance of this l2 with CNN model is evaluated on the test set, and the following visualisations are provided:

First, training and validation accuracy and loss curve:

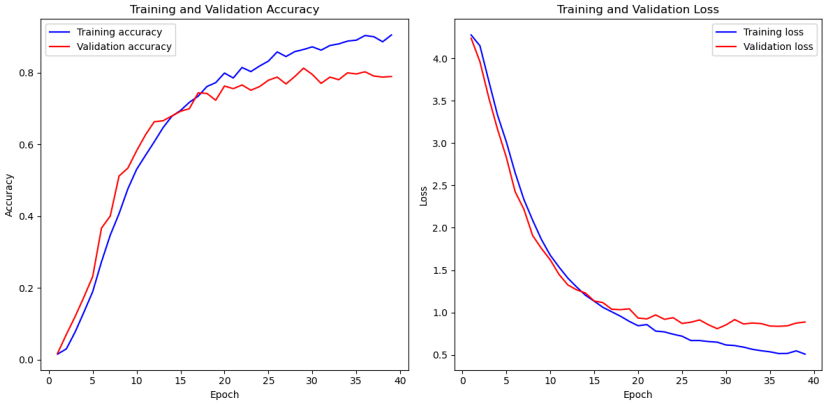
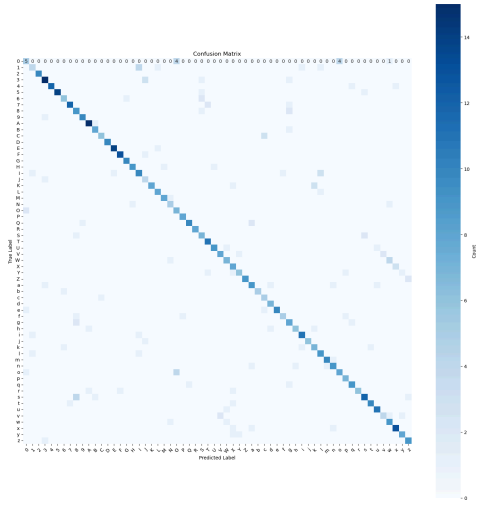


Figure 27. Loss and accuracy curves

Print the matrices and calculates various metrics to evaluate the model's performance:



accuracy			0.79	682
macro avg	0.80	0.79	0.79	682
weighted avg	0.80	0.79	0.79	682

Figure 28. Confusion matrix

With Higher Evaluation Accuracy: 80.21%

```
22/22 [==============================] - 2s 86ms/step - loss: 0.8264 - accuracy: 0.8021
Test Loss: 0.8263633251190186
Test Accuracy: 80.21%
```

Figure 29. Evaluation Accuracy

5.3.7 CNN with l2 model by using Hyperparameter Tuning

5.3.7.1 Model Architecture

The model starts with an input layer, which takes an image of shape input_shape. Then, a variable number of convolutional layers are added in a loop. The number of layers is a hyperparameter, ranging from 1 to 5. Each convolutional layer has a number of filters and a kernel size (from 3 to 5), both of which are hyperparameters. The layers use 'relu' activation, 'same' padding, and L2 regularisation, with the regularisation rate being a hyperparameter. After each convolutional layer, a max pooling layer and a dropout layer are added. The dropout rate is also a hyperparameter.
The output of the last dropout layer is reshaped to prepare it for the GRU layer. The GRU layer has a number of units (from 64 to 256 with each step = 32), which is a hyperparameter. It uses L2 regularisation, with the regularisation rate being a hyperparameter, and is followed by a dropout layer, with the dropout rate being a hyperparameter.

Next, a fully connected layer is added, with the number of units and the L2 regularisation rate being hyperparameters. This layer is followed by another dropout layer, with the dropout rate being a hyperparameter. Finally, an output layer is added, which uses softmax activation to output a probability distribution over the classes.

The model is then compiled with the Adam optimizer, with the learning rate being a hyperparameter. The loss function is set to 'sparse_categorical_crossentropy', and the metric to monitor is 'accuracy'. The build method returns the compiled model, ready to be used for hyperparameter tuning.

5.3.7.2 Initialize the HyperModel and Tuner

Initialises a ICRHyperModel_regu object called hypermodel. It takes two arguments: input_shape and num_classes. Creates a RandomSearch tuner object named tuner. The maximum number of hyperparameter combinations to try during the search. It is set to 50.

5.3.7.3 Best Model Evaluation and Visualization

Get the best model hyperparameters after 25 trials, and build a model with best hyperparameters. Train the best model with 50 epochs and get the final evaluation result which is 78.15 %, accuracy and loss curve and confusion matrix.

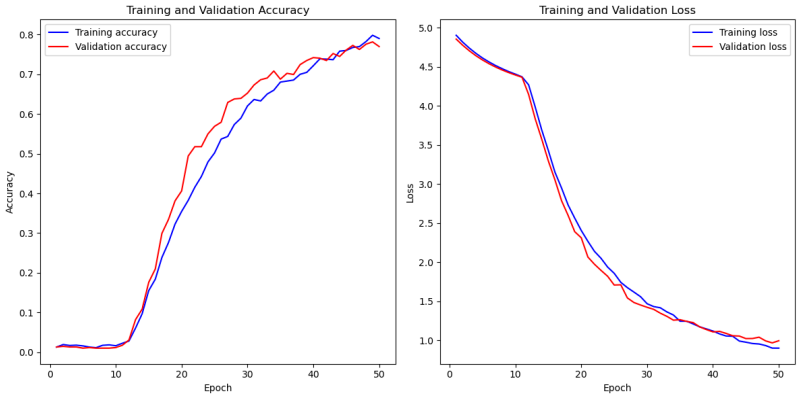
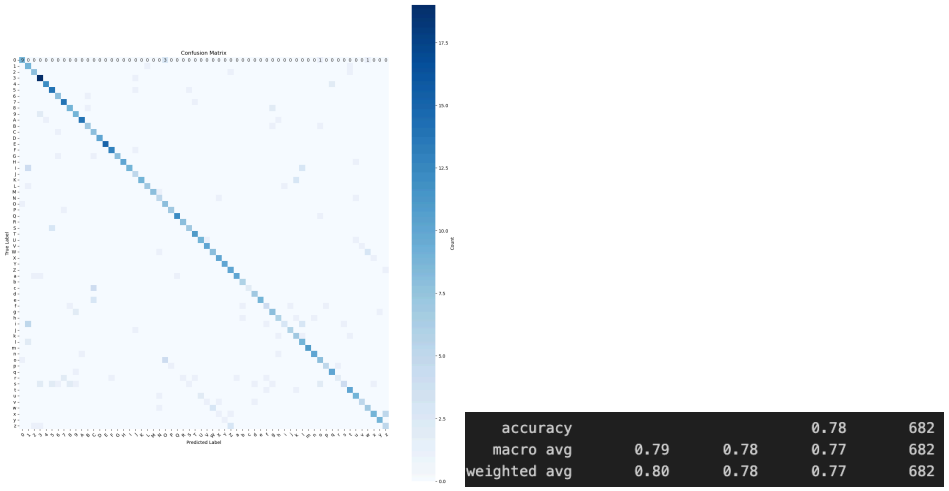


Figure 30. Accuracy and loss curve



accuracy			0.78	682
macro avg	0.79	0.78	0.77	682
weighted avg	0.80	0.78	0.77	682

Figure 31. confusion matrix

5.3.8 CNN Model with Long short-term memory (LSTM)

Though the accuracy of both CNN hyperparameter models is currently rather good, I discovered other methods that work well in handwriting recognition, such as LSTM which can recognise handwriting by learning the links between handwriting pictures and the text they relate to [35], and I'd want to try CNN with the LSTM model as well.

Long Short-Term Memory (LSTM) is a form of Recurrent Neural Network (RNN) that can maintain long-term dependencies in sequential input. LSTMs can process and analyse

sequential data, including time series, text, and voice. They manage the flow of information using a memory cell and gates, allowing them to selectively keep or discard information as required while avoiding the vanishing gradient problem that affects typical RNNs [35].

The model entails constructing three CNN architectures with 3, 4, and 5 convolutional layers, followed by an LSTM layer for image classification. Starting with an initial 2D convolutional layer comprising 32 filters, subsequent layers include sets of convolutional and max-pooling operations, each with 64 filters, to progressively extract more complex features. Dropout layers are integrated for regularisation, and the output is reshaped to align with the LSTM layer, tailored to capture temporal dependencies. Finally, a fully connected layer with softmax activation is appended for the ultimate classification. Although the model with 5 convolutional layers is speculated to potentially yield superior performance due to its enhanced feature extraction capabilities, empirical validation is essential to ascertain the optimal architecture for the classification task. For further elaboration on the model architecture, compilation, and training procedures, please consult Appendix XI for comprehensive details and instructions.

The performance of this LSTM with CNN model is evaluated on the test set, and the following visualisations are provided:

Training and validation accuracy and loss are monitored and visualised model performance.

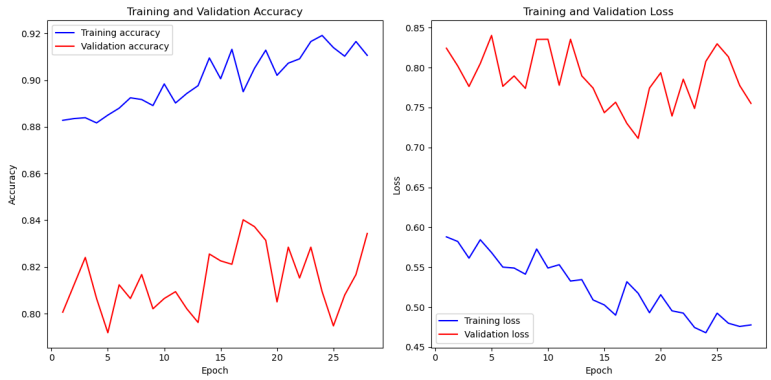
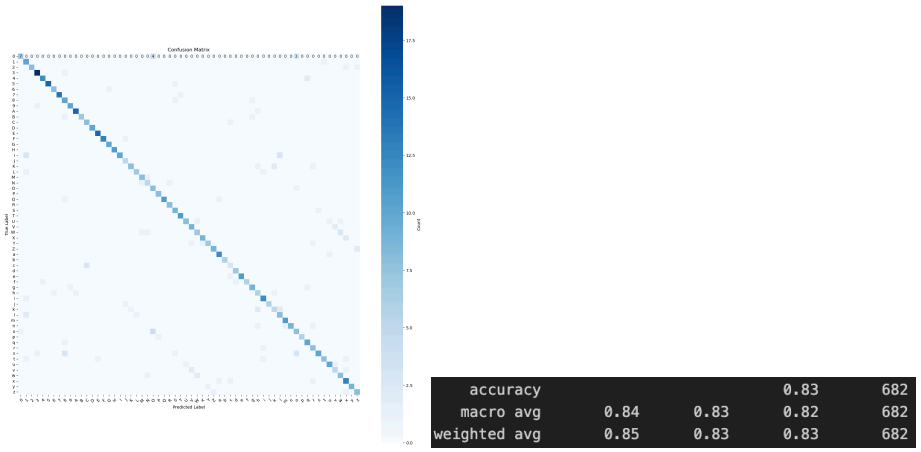


Figure 32. Loss and accuracy curves

Print the classification_report(y_val_encoded, y_pred_labels) matrices to evaluate the model's performance.



accuracy			0.83	682
macro avg	0.84	0.83	0.82	682
weighted avg	0.85	0.83	0.83	682

Figure 33. Confusion matrix

With Higher Evaluation Accuracy:82.99%

```
22/22 [==============================] - 1s 39ms/step - loss: 0.7288 - accuracy: 0.8299
Test Loss: 0.728795051574707
Test Accuracy: 82.99%
```

Figure 34. Evaluation Accuracy

5.3.9 Best Hyperparameter CNN with l2 model by using Median Filter

As I explained before in the Chinese Character model, the dataset source can also have an impact on the model's accuracy. So filters can also assist to improve model correctness in some manner. Here, I applied the median filter again to the original datasets to check if it improved model accuracy.

5.3.9.1 Apply Median Filter to Images

apply_median_filter function is using a built-in function called median_filter to apply the filter to the input image. The median_filter function takes two arguments: the image to be filtered and the size of the filter. In this case, the filter size is set to 3.

Apply the defined median filter function by using .flow function in ImageDataGenerator class to both X_train, y_train_encoded and X_test, y_test_encoded, and get train_generator_median_filter and test_generator_median_filter.

5.3.9.2 Model Training and Compilation

Load the saved best hyperparameter CNN with the l2 model from the desktop. Define the early stopping and reduce learning rate callbacks. Compile the model with configurations, optimiser, loss function and metrics. Fit the model on the training data as the previous CNN model.

5.3.9.3 Model Evaluation and Visualization

Training and validation accuracy and loss to assess model performance.

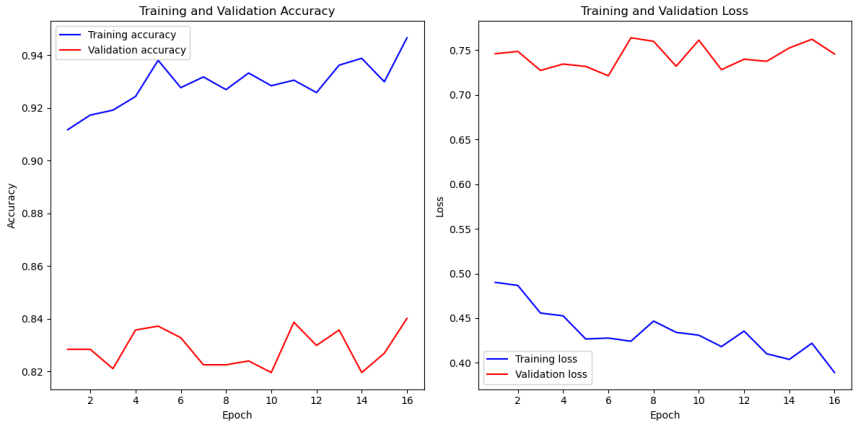
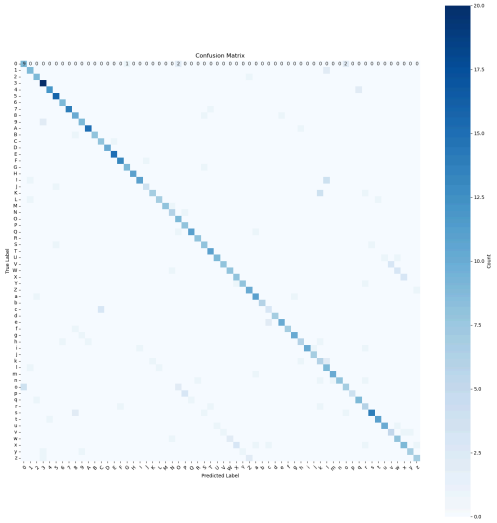


Figure 35. Loss and accuracy curves

Print the classification_report(y_val_encoded, y_pred_labels) matrices to evaluate the model's performance.



accuracy			0.82	682
macro avg	0.83	0.82	0.82	682
weighted avg	0.84	0.82	0.82	682

Figure 36. Confusion matrix

With Higher Evaluation Accuracy:84.02%

```
22/22 [==============================] - 2s 82ms/step - loss: 0.7253 - accuracy: 0.8402
Test Loss: 0.7253147959709167
Test Accuracy: 84.02%
```

Figure 37. Evaluation Accuracy

5.3.10 Summary

By employing various methods for training English character recognition models, including CNN with GRU, CNN with L2 Regularisation, CNN with LSTM with different numbers of convolutional layers, training epochs, and hyperparameter tuning, I yielded diverse accuracy scores. Specifically, the CNN with GRU model achieved 69.65% accuracy, while incorporating ImageDataGenerator improved it to 71.41%. Hyperparameter tuning for the CNN with GRU model further boosted the accuracy to 75.22%. The CNN with L2 Regularisation model performed better, reaching 80.21% accuracy, and its hyperparameter-tuned variant attained 78.15%. Regarding the CNN with LSTM models, the one with 3 convolutional layers achieved 79.33% accuracy, and model with 4 convolutional layers with accuracy 81.67%, while the model with 5 convolutional layers and 50 training epochs achieved 81.09% accuracy, further improving to 82.99% when trained for 100 epochs. Notably, applying a median filter to the

best-performing hyperparameter-tuned CNN with L2 Regularisation model yielded the highest accuracy of 84.02%.

5.4 Korean Character

5.4.1 Data Loading and Preprocessing

The Korean character recognition task utilises a dataset consisting of images of Korean characters, each labelled with the corresponding character. A custom function `process_images` is employed to preprocess the images. This function converts the images to grayscale, resizes them to 64x64 pixels, and flattens the image data into a one-dimensional array. The preprocessed images and their labels are stored in a created `DataFrame`, which is then exported as a CSV file for easy access and manipulation.

5.4.2 Exploratory Data Analysis (EDA) & Label Encoding

The pixel values in the dataset are converted to numeric format, with appropriate error handling mechanisms in place. The character labels are encoded into integers using the `LabelEncoder` from the scikit-learn library. An exploratory data analysis is performed to understand the distribution of different character classes in the dataset.

5.4.3 Train-Test Split

The dataset is split into training and testing sets using the `train_test_split` function from scikit-learn. To maintain class balance, stratification is applied during the splitting process. The pixel values are normalised, and the data is reshaped to fit the input requirements of a convolutional neural network (CNN).

5.4.4 Data Preprocessing and Augmentation

Getting data ready for training machine learning models, especially deep learning models like convolutional neural networks (CNNs), requires essential preprocessing and augmentation steps. These techniques convert the data into an appropriate format for the model to learn from effectively while also enhancing the model's ability to generalise its learning to new, unseen data.

One essential preprocessing step is one-hot encoding of categorical labels. Many machine learning models, including CNNs, require the labels to be represented as binary vectors, where each element corresponds to a specific class. One-hot encoding transforms categorical labels into these binary vectors, enabling the model to understand the categorical nature of the data and make accurate predictions. The labels (y_train and y_test) are one-hot encoded using tf.keras.utils.to_categorical to convert them into a suitable format for the model's output layer.

Another crucial preprocessing step is reshaping the input data to match the expected input shape of the CNN model. CNNs typically expect input data in the form of a 4D tensor, where

the dimensions represent the number of samples, image height, image width, and the number of channels. Reshaping the input data to match this expected shape ensures that the model can correctly process the input and learn meaningful representations. The input datasets (X_train and X_test) are reshaped to match the expected input shape of the CNN model, which is (number of samples, 64, 64, 1) for grayscale images with a height and width of 64 pixels.

One widely used approach for data augmentation is the ImageDataGenerator class in TensorFlow/Keras. An ImageDataGenerator is created with specified data augmentation parameters (width_shift_range, height_shift_range, shear_range, and zoom_range). This will apply various transformations to the input images during training, which can help improve the model's generalisation performance.

Additionally, learning rate scheduling is a technique that can help optimise the training process by adjusting the learning rate dynamically. The learning rate is a crucial hyperparameter that determines the step size of the updates applied to the model's weights during training. A well-designed learning rate schedule can improve convergence and prevent the model from getting stuck in suboptimal solutions. A learning rate scheduler function (lr_schedule) is defined, which decreases the learning rate by a factor of 0.5 every 5 epochs. This can help the model converge more effectively during training. A LearningRateScheduler callback is created using the lr_schedule function, which will be passed to the fit function of the model to adjust the learning rate during training.

By combining these data preprocessing and augmentation techniques, deep learning models like CNNs can effectively learn from the available data and achieve improved performance and generalisation capabilities. These techniques are particularly valuable when working with limited or imbalanced datasets, as they can help maximise the information extracted from the available data and mitigate potential overfitting issues.

5.4.5 Basic CNN Model

A CNN model is constructed using the Sequential API from Keras. The model architecture consists of convolutional layers followed by max-pooling layers, a flattening layer, and dense layers. The last layer utilises a softmax activation function for multi-class classification. The model is compiled with the Adam optimizer and categorical cross-entropy loss function.

The model architecture follows a structured progression from image input to final classification, embodying the fundamental principles of convolutional neural networks (CNNs). Beginning with a standard input layer tailored for 62x62 pixel RGB images, the model initiates feature extraction via a 2D convolutional layer equipped with 32 filters, detecting basic features like edges and colours. Subsequent MaxPooling2D layers reduce spatial dimensions, enhancing computational efficiency and introducing translational

invariance. Transitioning to classification, a Flatten layer reshapes the feature maps for dense layer processing, facilitating the integration of learned features. Dense layers further refine feature integration, culminating in a final dense layer tailored to the specific classification task, in this case, 30 output units representing class probabilities. Through a sequential arrangement of layers, the model seamlessly progresses from feature extraction to classification, embodying the core principles of CNN architectures. For additional insights into the model architecture, please refer to Appendix X for a detailed breakdown and explanation.

The performance of the basic CNN model is evaluated on the test set, and the following visualisations are provided:

A. Loss and Accuracy Curves: Plots showing the training and validation loss, as well as the training and validation accuracy over the training epochs.

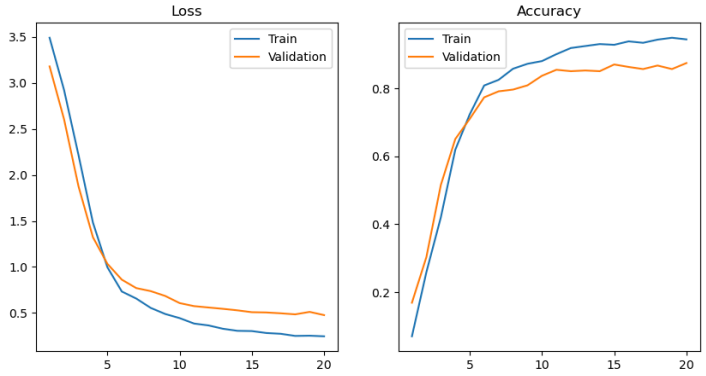


Figure 38. Loss and Accuracy Curves

B. Confusion Matrix: A graphical representation of the model's predictions compared to the true labels, providing insights into the performance of the model for each character class.

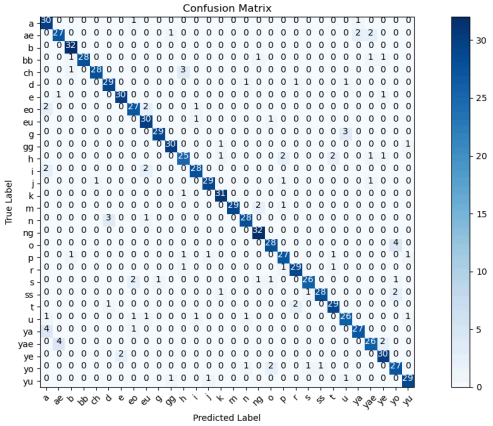


Figure 39. Confusion Matrix

The overall accuracy of the model is 88.96% which is reported on the test set.

Test Loss: 0.4286452829837799

Test Accuracy: 88.96%

5.4.6 Hyperparameter Tuning

To improve the model's performance, hyperparameter tuning is performed using Keras Turner's RandomSearch. This process involves systematically searching for the optimal configuration of various hyperparameters, such as the number of convolutional layers, filter sizes, kernel sizes, and the number of units in dense layers. After conducting 15 trials, the accuracy is improved to 98%.

The best hyper model architecture for image classification presented in section 5.4.6.1 embodies a meticulously crafted journey from input image processing to final classification. Starting with an implicit input layer capable of handling images slightly larger than 64x64 pixels, the model's deep feature extraction begins with a Conv2D layer equipped with 96 filters, detecting a diverse range of features from simple edges to complex patterns. Alternating between convolutional and max pooling layers, the model progressively refines feature maps, culminating in a distilled essence of the input image. Transitioning to classification, a Flatten layer converts the feature maps into a 1D vector, ready for interpretation by dense layers. A dense layer with 384 units integrates the learned abstract representations, serving as the backbone for classification, while the final dense output layer with 30 units translates integrated features into probabilities across 30 classes. This architecture optimally balances feature extraction and computational efficiency, making it highly adept for complex image classification tasks. For more details about the model architecture please refer to Appendix X.

After hyperparameter tuning, the performance of the trained model is evaluated on the test set, and the results are printed. Additionally, the following visualisations are provided:

Training and Validation Loss and Accuracy Curves:

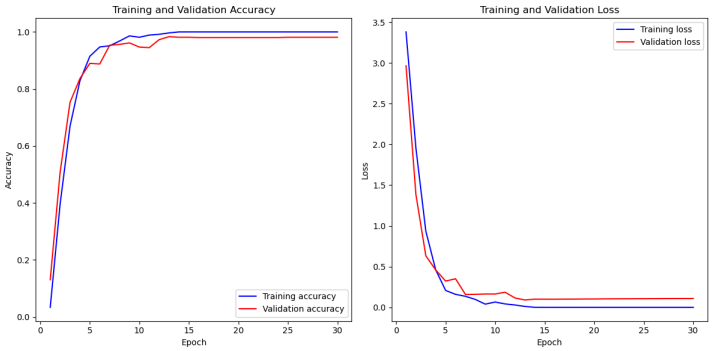


Figure 40. Loss and Accuracy Curves

Confusion Matrix:

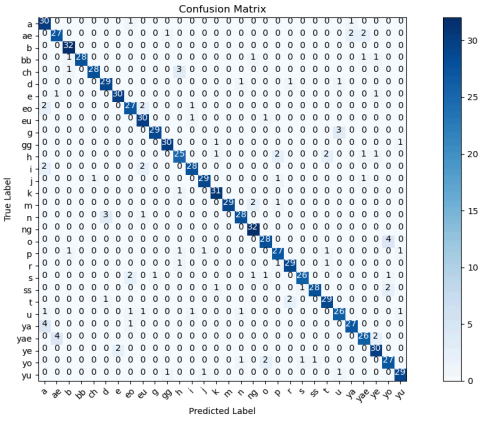


Figure 41. Confusion Matrix

The overall accuracy of the model is 98.12% which is reported on the test set.

Test Loss: 0.10897793620824814

Test Accuracy: 98.12%

5.4.7 Summary

By employing Keras Tuner's RandomSearch and systematically exploring different hyperparameter configurations, the first model was able to achieve a significantly higher accuracy of 98%, outperforming the basic CNN model's accuracy of 88.96%.

Chapter 6: IMPLEMENTATION

Loading the pre-trained Chinese, English and Korean Character model from a directory specified by `model_directory`. The `tf.keras.models.load_model` function is used to load the trained model.

6.1 Implementation of Chinese Character Model

The `load_class_names` function begins by compiling a regular expression pattern, which matches characters followed by indices, allowing for potential whitespace between them. It then opens the specified file in read-only mode with UTF-8 encoding to ensure proper handling of the content. Within a `with` statement, the function reads the entire file content. Using the `findall` method on the compiled pattern, it extracts all matches in the file content, returning a list of tuples containing matched characters and their respective indices. The function computes the maximum index by extracting indices from the matches and finding the maximum value using a generator expression with `max`. Subsequently, it initialises a list named `class_names` with `None` placeholders, ensuring its size is sufficient to accommodate all class names at their respective indices. Finally, iterating over matches, it assigns each character to the appropriate index in `class_names`, converting the index to an integer for indexing. The function ultimately returns `class_names`, which contains the class names ordered by their indices.

The `median_filter_wrap` function is a wrapper around the `median_filter` function from the `scipy.ndimage` library. It expects a NumPy array as input and applies a median filter with a kernel size of 3 to the image. This function is used to preprocess the input image by reducing noise.

The `apply_median_filter` function is a helper function that uses `tf.py_function` to apply the `median_filter_wrap` function to the input image tensor. `tf.py_function` is used to execute Python functions within the TensorFlow graph. The function ensures that the output tensor has the correct shape after applying the median filter.

The `preprocess_image` function is responsible for preprocessing the input image tensor before feeding it into the model. It resizes the image to a target size of (64, 64) pixels. If the input image is a colour image (3 channels), it converts it to grayscale. It applies the median filter to the grayscale image using `apply_median_filter`. The image is then normalised by dividing by 255.0. Finally, it adds a batch dimension to the image tensor using tf.expand_dims to match the expected input shape of the model.

The `predict_character` function takes the preprocessed image tensor as input. It passes the tensor through the loaded model using `loaded_model.predict`. The predicted class index with the highest probability is obtained using `np.argmax`. The predicted character is retrieved from the `class_names` list using the predicted index. The function returns the predicted Chinese character.

6.2 Implementation of English Character Model

The `load_and_preprocess_image` function is defined to preprocess the input image tensor. It resizes the image to a target size of (128, 128) pixels using `tf.image.resize`. The image is then normalised by dividing by 255.0. The function returns the normalised image tensor.

Defining the `labels` dictionary, which maps the class indices (0 to 61) to their corresponding English characters (0-9, A-Z, a-z).

The `predict_character_english` function decodes an uploaded file into an image tensor with 3 channels, assuming it's a colour image, using `tf.io.decode_image`. Subsequently, the image tensor undergoes preprocessing, likely involving resizing, normalisation, or other transformations, via a function named `load_and_preprocess_image`, presumably defined elsewhere. Following preprocessing, a batch dimension is added to the tensor using `tf.expand_dims` to accommodate model expectations. The preprocessed image tensor is then passed to `model.predict` for character predictions, where `model` is assumed to be a trained machine learning model capable of character prediction. The predictions, likely in the shape (1, num_classes), where `num_classes` denotes the number of characters the model predicts, are stored. The function retrieves the index of the highest probability

prediction using `np.argmax(predictions, axis=1)[0]`, followed by fetching the corresponding character from an unspecified `labels` list. This character is returned as the function's output.

6.3 Implementation of Korean Character Model

The `preprocess_image` function is updated to handle grayscale conversion and normalisation for the Korean character recognition model. It opens the input image using `Image.open` from the `PIL` library and converts it to grayscale using `convert('L')`. The grayscale image is resized to (64, 64) pixels using `image_pil.resize((64, 64))`. The resized image is converted to a NumPy array and normalised by dividing by 255.0. A channel dimension is added to the grayscale image using `np.expand_dims(image_np, axis=-1)`. A batch dimension is added using `np.expand_dims(image_np, axis=0)`. The preprocessed and batched image tensor is returned.

Defining the `dic` dictionary, which maps the class indices to their corresponding Korean characters and their romanized representations. Each key in the dictionary is an integer index, and the value is a tuple containing the romanized form and the Korean character.

The `predict_korean_character` function takes an input image as input. It loads the pre-trained Korean character recognition model using `tf.keras.models.load_model`. The input image is preprocessed using the `preprocess_image` function. The preprocessed image tensor is passed through the loaded model using `loaded_model_korean.predict` to obtain the predictions. The predicted class index with the highest probability is obtained using `np.argmax(predictions)`. The predicted character is retrieved from the `dic` dictionary using the predicted index. The function returns the predicted Korean character and its romanized form as a tuple.

Sets up a Streamlit app for uploading an image of a Chinese, English or Korean character. The user can upload an image file of type PNG, JPG, or JPEG using the `st.file_uploader` function. If an image is uploaded, the code calls the `predict_character_chinese` `predict_character_english` or `predict_korean_character` function with the uploaded file. The predicted Chinese, English or Korean character and its romanized form are displayed using `st.write`.

Chapter 7: EVALUATION AND FUTURE WORKS

7.1 Evaluation

The current approach has tackled the recognition of Chinese characters, but there are inherent challenges due to the vast quantity and complexity of the Chinese writing system. With over 50,000 characters in the complete set, this project focused on a subset due to GPU limitations. The diversity and visual similarity among Chinese characters pose significant

difficulties for accurate recognition. While the models achieved reasonable performance on English character recognition, the accuracy levels are still not satisfactory for practical applications. The highest accuracy obtained was 84.02% using a hyperparameter-tuned CNN with L2 regularisation and median filtering. However, this accuracy may not be sufficient for scenarios requiring high precision, such as document digitization or language learning. The approach demonstrated excellent performance on the Korean character dataset, achieving an accuracy of 98% with a hyperparameter-tuned model. However, it is important to note that the dataset used consisted of only 29 array elements representing Korean characters. In practice, Korean writing combines these elements into more complex character formations, which were not addressed in this project. Unfortunately, suitable datasets for Japanese character recognition were not available during this project, resulting in a lack of evaluation and development for this writing system.

The current models are limited to recognizing individual characters. However, in real-world scenarios, texts often appear as sentences or paragraphs, requiring the ability to recognize and segment individual characters from continuous text.

7.2 Future Work

To enhance Chinese character recognition, acquiring a more comprehensive dataset covering a broader range of characters and writing styles is crucial for improved representation and generalisation. Optimising models for English character recognition requires exploring advanced architectures, attention mechanisms, and transfer learning techniques. Extending Korean character recognition involves acquiring datasets with compound character formations and developing models capable of segmenting individual character elements. Acquiring suitable datasets is essential for future Japanese character recognition efforts, enabling the development of specialised models and techniques tailored to its complexities.

Extending the models to recognize and segment characters from continuous text, such as sentences or paragraphs, would greatly enhance the practical applicability of the character recognition approach. This could involve integrating techniques like sequence labelling, language modelling, or attention-based mechanisms to capture contextual information and improve recognition accuracy.

By addressing these limitations and focusing on the proposed future work, the character recognition approach can be further enhanced, enabling more robust and practical applications across diverse writing systems and real-world scenarios.

Chapter 8: SUMMARY AND REFLECTIONS

8.1 SUMMARY

Throughout this project, my deep familiarity with Chinese characters provided a strong foundation for exploring character recognition across diverse writing systems, including English, Korean, and the potential inclusion of Japanese. Leveraging various models and techniques, we aimed to develop robust solutions tailored to each writing system's unique challenges.

In Chinese character recognition, the implementation of data augmentation techniques, notably the median filter, significantly improved accuracy by up to 4%. Additionally, the integration of additional convolutional layers within the CNN architecture yielded notable performance enhancements, achieving an accuracy of 82.7%. However, the intricate nature of the Chinese writing system, characterised by a vast character repertoire and visual intricacies, presented considerable hurdles.

In the realm of English character recognition, combining recurrent neural networks (RNNs) such as GRU and LSTM with CNN architectures showcased promising results. Techniques like L2 regularisation, hyperparameter tuning, and median filter application contributed to accuracy improvements, with the top-performing model achieving an accuracy of 84.02%. Yet, further refinements are necessary to meet the stringent precision requirements of real-world applications.

The approach exhibited remarkable accuracy on the Korean character dataset, achieving 98% accuracy with a hyperparameter-tuned model. However, it's essential to acknowledge the dataset's limitations, consisting of only 29 array elements representing Korean characters. Future endeavours should focus on recognizing compound character formations, aligning with Korean writing conventions.

Regrettably, due to the unavailability of suitable datasets, Japanese character recognition remained unexplored in this project, underscoring the imperative for data acquisition and development in this domain.

In summary, while my expertise in Chinese characters provided a strong starting point, the project's methodologies and insights extend to character recognition across various writing systems. By employing tailored models and techniques and addressing unique challenges, we aspire to develop versatile and effective character recognition solutions applicable across diverse linguistic contexts.

8.2 REFLECTIONS

Reflecting on the project's limitations, one notable aspect is the focus on single character recognition. While this is a crucial step, practical applications often require the recognition and segmentation of characters from continuous text, such as sentences or paragraphs. Extending the models to handle this aspect would significantly enhance their applicability and usefulness.

Overall, this project has made significant strides in developing character recognition capabilities across multiple writing systems. However, several challenges and areas for improvement remain, particularly in addressing the complexities of Chinese and Japanese writing systems, further enhancing recognition accuracy, and expanding the scope to accommodate continuous text recognition.

Future efforts should focus on acquiring comprehensive datasets and exploring advanced model architectures, including attention mechanisms and techniques like transfer learning. Collaboration within the research community and standardisation of evaluation protocols would further drive progress in character recognition. Ultimately, accurate recognition systems have broad applications, from digitising documents to promoting cultural preservation and accessibility.

Chapter 9: Reference

[1] Wikipedia contributors, 2021. Intelligent character recognition. Wikipedia, The Free Encyclopedia, (27 January 2021). [Accessed 19 October 2023].
[2] Felkel, R., 2021. WHAT YOU NEED TO KNOW ABOUT OCR AND ICR TECHNOLOGIES. [Accessed 19 October 2023].
[3] Tripathi, P., 2021. What is Intelligent Character Recognition?. [Accessed 19 October 2023].
[4] Taylor, K., 2017. OCR vs ICR: What Differentiates the two Character Recognition Software?. [Accessed 19 October 2023].
[5] IBM, n.d. "What is machine learning?." Available at: https://www.ibm.com/topics/machine-learning [Accessed 19 October 2023].
[6] Ptucha, R., Such, F.P., Pillai, S., Brockler, F., Singh, V., and Hutkowski, P., n.d. Intelligent character recognition using fully convolutional neural networks. Pattern Recognition. Available at: https://doi.org/10.1016/j.patcog.2018.12.017 [Accessed 19 October 2023].
[7] Vishnuvardhan, G., Ravi, V., and Santhosh, M., 2021. Intelligent Character Recognition with Shared Features from Resnet. In: Gunjan, V.K., and Zurada, J.M. (eds) Modern Approaches in Machine Learning and Cognitive Science: A Walkthrough. Studies in Computational Intelligence, vol 956. Springer, Cham. Available at: https://doi.org/10.1007/978-3-030-68291-0_21 [Accessed 19 October 2023].
[8] C. Wigington et al., “Start, follow, read: End-to-end full page handwriting recognition,” Computer Vision Foundation, https://openaccess.thecvf.com/content_ECCV_2018/papers/Curtis_Wigington_Start_Follow_Read_ECCV_2018_paper.pdf [accessed Oct. 20, 2023].
[9] A. S. Gillis, E. Burns, and K. Brush, “What is deep learning and how does it work?: Definition from TechTarget,” Enterprise AI, https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network [accessed Mar. 15, 2024].
[10] Cheriet, M., Nawaz, N., & Arora, S. (2009). Optical character recognition: An illustrated guide to the frontier. Springer Science & Business Media [accessed Mar. 16, 2024].
[11] Govindan, V. K., & Shivaprasad, A. P. (1990). Character recognition - A review. Pattern Recognition, 23(7), 671-683 [accessed Mar. 16, 2024].

[12] Pal, U., & Chaudhuri, B. B. (2004). Indian script character recognition: A survey. Pattern Recognition, 37(9), 1887-1899 [accessed Mar. 16, 2024].
[13] Plamondon, R., & Srihari, S. N. (2000). Online and off-line handwriting recognition: A comprehensive survey. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(1), 63-84 [accessed Mar. 16, 2024].
[14] Yuan Zhuang et al 2021 J. Phys.: Conf. Ser. 1820 012162 A Handwritten Chinese Character Recognition based on Convolutional Neural Network and Median Filtering [accessed Mar. 16, 2024].
[15] Team, G.L. (2023) *Label encoding in python - 2024*, *Great Learning Blog: Free Resources what Matters to shape your Career!* Available at: https://www.mygreatlearning.com/blog/label-encoding-in-python/#:~:text=Label%20encoding%20is%20a%20technique,only%20operate%20on%20numerical%20data [Accessed: 27 March 2024].
[16] Saturn Cloud (2023) Why you need to compile your keras model before using model.evaluate(), Saturn Cloud Blog. Available at: https://saturncloud.io/blog/why-you-need-to-compile-your-keras-model-before-using-modelevaluate/#:~:text=In%20Keras%2C%20compiling%20the%20model,to%20minimize%20the%20loss%20function. [Accessed: 27 March 2024].
[17] *What is model training* (2023) *Iguazio*. Available at: https://www.iguazio.com/glossary/model-training/ [Accessed: 27 March 2024].
[18] AWS *What is hyperparameter tuning? - hyperparameter tuning methods ...* Available at: https://aws.amazon.com/what-is/hyperparameter-tuning/#:~:text=Hyperparameters%20directly%20control%20model%20structure,values%20is%20crucial%20for%20success. [Accessed: 27 March 2024].
[19] Ramuglia, G. (2024) *Using train test split in Sklearn: A complete tutorial*, *Linux Dedicated Server Blog*. Available at: https://ioflood.com/blog/train-test-split-sklearn/#:~:text=train_test_split%20is%20not%20just%20for,a%20single%20scikit%2Dlearn%20estimator. [Accessed: 27 March 2024].
[20] Hariharan, P. (2024) *Numpy reshape() - function for reshaping arrays*, *Flexiple*. Available at: https://flexiple.com/python/numpy-reshape# [Accessed: 27 March 2024].
[21] AWS *What is data augmentation? - data augmentation techniques explained ...* Available at: https://aws.amazon.com/what-is/data-augmentation/#:~:text=Data%20augmentation%20is%20the%20process,machine%20learning%20(ML)%20models. [Accessed: 27 March 2024].
[22] Yamashita, R. *et al.* (2018) *Convolutional Neural Networks: An overview and application in radiology - insights into imaging*, *SpringerOpen*. Available at: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9#Sec4 [Accessed: 27 March 2024].
[23] Ashhar , S.M. (2024) *What is a neural network flatten layer?*, *Educative*. Available at: https://www.educative.io/answers/what-is-a-neural-network-flatten-layer# [Accessed: 27 March 2024].
[24] Wei, D. (2024) *Demystifying the adam optimizer in machine learning*, *Medium*. Available at: https://medium.com/@weidagang/demystifying-the-adam-optimizer-in-machine-learning-4401d162cb9e [Accessed: 30 March 2024].
[25] Rahman, M. (2023) *What you need to know about sparse categorical cross entropy*, *Medium*. Available at: https://rmoklesur.medium.com/what-you-need-to-know-about-sparse-categorical-cross-entropy-9f07497e3a6f#:~:text=Sparse%20Categorical%20Cross%20Entropy%20is,used%20for%20binary%20classification%20problems. [Accessed: 30 March 2024].
[26] D, E. (2019) *Accuracy, Recall & Precision*, *Medium*. Available at: https://medium.com/@erika.dauria/accuracy-recall-precision-80a5b6cbd28d [Accessed: 30 March 2024].
[27] Shukla, D. (2024) *All about init in Python*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2024/02/all-about-init-in-python/# [Accessed: 01 April 2024].
[28] Biswal, A. (2024) *The complete guide on overfitting and underfitting in machine learning*, *Simplilearn.com*. Available at: https://www.simplilearn.com/tutorials/machine-learning-tutorial/overfitting-and-underfitting [Accessed: 01 April 2024].
[29] Makoto Ohki, Michael E. Zervakis, Anastasios N. Venetsanopoulos, 3-D Digital Filters, Editor(s): C.T. Leondes, Control and Dynamic Systems, Academic Press, Volume 69, 1995, Pages 49-88, ISSN 0090-5267, ISBN 9780120127696, https://doi.org/10.1016/S0090-5267(05)80038-6.
(https://www.sciencedirect.com/science/article/pii/S0090526705800386)[Accessed: 01 April 2024].
[30] BAJWA, A. (no date) *What is Pandas dataframe.iterrows() in python?*, *Educative*. Available at: https://www.educative.io/answers/what-is-pandas-dataframeiterrows-in-python [Accessed: 02 April 2024].
[31] Anishnama (2023) *Understanding gated recurrent unit (GRU) in deep learning*, *Medium*. Available at: https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2 [Accessed: 02 April 2024].

[32] Team, K. (no date) *Keras Documentation: The base tuner class*, *Keras*. Available at: https://keras.io/api/keras_tuner/tuners/base_tuner/#get_best_hyperparameters-method [Accessed: 03 April 2024].
[33] Otten, N.V. (2023) *L1 and L2 regularization explained, when to use them & practical how to examples*, *Spot Intelligence*. Available at: https://spotintelligence.com/2023/05/26/l1-l2-regularization/ [Accessed: 03 April 2024].
[34] Cohen, J. (2021) *Recurrent neural networks (RNNS) in Computer Vision: Image captioning*, *Medium*. Available at: https://heartbeat.comet.ml/recurrent-neural-networks-rnns-in-computer-vision-image-captioning-ea9d568e0077 [Accessed: 08 April 2024].
[35] Banoula, M. (2023) *Introduction to long short-term memory(lstm): Simplilearn*, *Simplilearn.com*. Available at: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/lstm#:~:text=LSTMs%20Long%20Short%2DTerm%20Memory,series%2C%20text%2C%20and%20speech. [Accessed: 08 April 2024].

Chapter 10: Appendix

Appendix I

Data Loading and Preprocessing

The dataset is stored in a specific directory structure, where each folder represents a character class, and the images corresponding to that character are stored within. Initialise lists `images = []` and `labels = []` to store the images and labels.

A function named `preprocess_image` is created to perform several tasks: converting each image to grayscale, resizing it to 64x64 pixels, and normalising pixel values within the range of 0 to 1. Another function, `get_label_from_folder_name`, is defined to extract labels from folder names. The `process_images` function is used to iterate through each character's folder, applying the `preprocess_image` function to process each image, while simultaneously extracting the corresponding label. These processed images and their respective labels are then added separately to lists in the correct order. Checks are included to manage situations where data is missing or invalid, and any errors that occur during image processing are displayed.

The `process_images` function is invoked for every character folder within the train_path directory. It retrieves the paths of the images, processes the images along with their labels using the `get_label_from_folder_name` function.

To enhance computational efficiency, ensure uniform data types, and facilitate integration with scikit-learn libraries for subsequent dataset splitting, the processed images and labels are transformed into NumPy arrays.

The `train_test_split` function serves a broader purpose beyond simple data division. It is a versatile tool applicable in extensive machine learning endeavours, including the construction of machine learning pipelines. Within a pipeline, it integrates seamlessly with other steps such as data preprocessing, model training, and model evaluation, consolidating them into a unified scikit-learn estimator [19]. The data is split into training and validation sets using `train_test_split` with a test size of 0.2 and a random state of 42.

The `numpy.reshape()` function in Python enables us to alter the shape of an array. Reshaping involves modifying the arrangement of elements within the array. The shape of an array is defined by the count of elements in each dimension. Through reshaping, we have the flexibility to add or remove dimensions from the array. Additionally, we can adjust the quantity of elements within each dimension as needed [20]. This reshaping ensures that the datasets conform to the expected input format of (batch_size, height, width, channels) for a

CNN model. By structuring the data in this manner, the code facilitates compatibility with the CNN, enabling it to efficiently process the images and extract relevant features.

Reshaping the X_train and X_val datasets is essential to ready the data for training the Chinese character CNN model later on. The `reshape` function is employed to alter the shape of X_train and X_val datasets, appending an additional dimension of 1 at the end to accommodate grayscale images. This is achieved by utilising -1 as the first dimension, indicating that its size is inferred based on the other dimensions. Specifically, the first -1 implies that the dimension's size is inferred from the array's length and the sizes of the other dimensions. The subsequent dimensions are denoted as X_train.shape[1] and X_train.shape[2], representing the height and width of each image, respectively. The 1 at the end signifies that each pixel value is represented as a single channel, which is standard for grayscale images.

Label Encoding

Since in my datasets the folder names representing the character classes are used as labels, which means each character class is associated with a specific folder, and the folder name serves as the label for that class.

A. Training Label Encoding
    a. The `fit_transform()` method is applied to the training labels (`y_train`) using the `LabelEncoder` instance.
    b. This method first learns the mapping between the unique character labels in `y_train` and assigns a unique numerical value to each label.
    c. After learning the mapping, it transforms the training labels into their corresponding numerical representations.
    d. The transformed numerical labels are stored in `y_train_encoded`.

B. Validation Label Encoding
    a. The `transform()` method is applied to the validation labels (`y_val`) using the same `LabelEncoder` instance that was fitted on the training labels.
    b. This method applies the mapping learned from the training labels to transform the validation labels into numerical representations.
    c. The transformed numerical labels are stored in `y_val_encoded`.

The purpose of using these encoding methods is to ensure that the categorical character labels are represented in a format that can be understood and processed by machine learning algorithms. By converting the labels into numerical form, the algorithms can perform mathematical operations on them and make predictions based on the encoded values.

After the encoding process, the preprocessed images (`images`) and their corresponding numerical labels (`y_train_encoded` and `y_val_encoded`) are used for training and evaluating the Chinese character recognition model.

Data Augmentation

Data augmentation involves creating additional data from existing datasets, primarily aimed at training new machine learning (ML) models. While ML models demand extensive and diverse datasets for effective training, acquiring such datasets can be difficult due to factors like data silos, regulations, and constraints. Data augmentation addresses this challenge by expanding the dataset through minor modifications to the original data. Recently, generative artificial intelligence (AI) solutions have emerged as effective means for rapidly and efficiently augmenting data across various industries, ensuring high-quality outcomes [21].

`datagen` is defined by using an instance of the `ImageDataGenerator` class from the Keras library which is commonly used for data augmentation in image classification tasks. The `datagen` object is configured with various transformation parameters: `width_shift_range=0.01`, `height_shift_range=0.01`, `shear_range=0.1`, and `zoom_range=0.1`. These parameters determine the scope of transformations applicable to input images during training.

1. width_shift_range and height_shift_range: These parameters regulate the extent of horizontal and vertical shifts in images, represented as a fraction of the total width or height.
2. shear_range: This parameter manages the intensity of shearing, which alters the shape of images.
3. zoom_range: It defines the span for randomly zooming in or out of images.

Generating train_generator and test_generator variables by using the flow method of the datagen object. And generators generate batches of augmented images and their corresponding labels on the fly, which can be fed into the training and testing processes of the model.

A. Train Generator
    a. Create a generator that will generate training data for the enhancement batch.
    b. The flow method takes the original training images (X_train) and their corresponding labels (y_train_encoded), along with additional parameters batch size (batch_size = 32) and shuffle the data (shuffle = False).
    c. Shuffle is set to False, meaning that the data won't be shuffled before being processed in batches.
B. Test Generator
    a. Creates a generator for augmented batches of validation data.

b. The flow method takes the original validation images (X_val) and their corresponding labels (y_val_encoded), along with similar parameters batch size (batch_size = 32) and shuffle the data (shuffle = False).
c. Shuffle is set to False, meaning that the data won't be shuffled before being processed in batches.

Data augmentation is crucial for enhancing the performance of deep learning models by enriching datasets with diverse variations of existing data, which allows models to encounter a broader range of features during training. This process not only improves the model's ability to generalise to unseen data but also reduces the dependency on large datasets, making training more efficient. Additionally, data augmentation helps mitigate overfitting by providing a more comprehensive dataset that prevents models from learning to rely too heavily on specific characteristics of the training data, thus improving the model's ability to make accurate predictions in real-world scenarios [21].

Appendix II

Model Architecture

A. Feature Extraction Layers
    a. Convolutional Layers
        i. `conv2d_3` (Conv2D): The first convolutional layer with 32 filters and a kernel size of 3x3. It processes the input image and extracts low-level features such as edges, lines, and curves.
        ii. `conv2d_4` (Conv2D): The second convolutional layer with 64 filters and a kernel size of 3x3. It extracts more complex features by combining the low-level features from the previous layer.
        iii. `conv2d_5` (Conv2D): The third convolutional layer with 64 filters and a kernel size of 3x3. It further combines and extracts higher-level features from the previous convolutional layer.
    b. Max-Pooling Layers
        i. `max_pooling2d_3` (MaxPooling2D): The first max-pooling layer with a 2x2 pool size is applied after the first convolutional layer. It helps in reducing the spatial dimensions of the feature maps and introduces translation invariance.
        ii. `max_pooling2d_4` (MaxPooling2D): The second max-pooling layer with a 2x2 pool size, applied after the second convolutional layer.
        iii. `max_pooling2d_5` (MaxPooling2D): The final max-pooling layer with a 2x2 pool size, applied after the third convolutional layer.

The convolutional layers and max-pooling layers work together to extract relevant features from the input images, progressively capturing more complex patterns and reducing spatial dimensions.

B. Integration and Classification Layers
    a. `flatten_1` (Flatten): This layer flattens the output from the convolutional and pooling layers into a one-dimensional vector, preparing the data for the fully connected layers.
    b. `dense_2` (Dense): The first fully connected or dense layer with 128 units. It combines the features extracted by the convolutional layers and learns higher-level representations.
    c. `dropout_1` (Dropout): A dropout layer with a rate of 0.2. It helps in reducing overfitting by randomly dropping out (setting to zero) a fraction of the input units during training.
    d. `dense_3` (Dense): The final fully connected layer with units equal to the number of unique labels or classes in the dataset. It performs the actual classification task by mapping the learned features to the corresponding class labels. This layer uses a softmax activation function for multi-class classification.

The integration and classification layers take the features extracted by the convolutional and pooling layers, integrate them, and make the final classification decision by mapping the learned representations to the corresponding character classes.

Model Compilation

A. `optimizer='adam'`: The optimizer to be used during training. The optimizer is responsible for updating the model's weights based on the calculated gradients, which helps the model learn and improve its performance.
B. `loss='sparse_categorical_crossentropy'`: The loss function to be used during training. The loss function measures how well the model is performing by comparing its predictions to the actual labels.
C. `metrics=['accuracy']`: The evaluation metrics to be used during training. The metrics provide additional information about the model's performance, but do not affect the training process itself. It is used to measure the proportion of correctly classified samples.

Model Training

A. The dataset is split into training and validation sets with an 80-20 ratio.
B. The model is trained for 10 epochs, with early stopping based on validation loss to prevent overfitting.

C. During training, the training and validation accuracy and loss are tracked and plotted for analysis.

Appendix III

Model Architecture

A. Defining a class called CustomCNN, which inherits from the models.Model class. This means the CustomCNN class is a subclass of the Keras Model class, which provides a set of methods and attributes for building and training neural network models.
B. Inside the CustomCNN class, there is an __init__ method. This method is called when an instance of the class is created. It takes an optional argument number_of_characters, which specifies the number of output units in the final dense layer of the model.
C. Within the __init__ method, the model architecture is defined. Here's a breakdown of the layers:
    a. Feature Extraction Layers
        i. Convolutional Layers
            1. `conv2d` (Conv2D): This is the first convolutional layer that applies 6 filters of size 3x3 to the input data, using the ReLU activation function. The padding='same' argument ensures that the output feature maps have the same spatial dimensions as the input.
            2. `conv2d_1` (Conv2D): This is the second convolutional layer that applies 12 filters of size 3x3 to the input feature maps, also using the ReLU activation function.
        ii. Max-Pooling Layers
            1. `max_pooling2d` (MaxPooling2D): This is the first max-pooling layer that performs max pooling with a pool size of 2x2, reducing the spatial dimensions of the feature maps by half.
            2. `max_pooling2d_1` (MaxPooling2D): This is the second max-pooling layer that performs another round of max pooling with a pool size of 2x2, further reducing the spatial dimensions.
    b. Integration and Classification Layers
        i. `flatten` (Flatten): This layer flattens the output feature maps from the convolutional and max-pooling layers into a one-dimensional vector, preparing the data for the final classification layer.
        ii. `dense` (Dense): This is the final fully connected or dense layer with number_of_characters units, where number_of_characters represents

the number of output classes or characters in the dataset. This layer uses the softmax activation function for multi-class classification, outputting the predicted probabilities for each character class.

The convolutional layers in the model are responsible for extracting low-level and high-level features from the input data, such as edges, shapes, and patterns present in the character images. The max-pooling layers help in reducing the spatial dimensions of the feature maps, effectively capturing the most salient features while providing translation invariance.
The flatten layer integrates the extracted features from the convolutional and max-pooling layers into a single vector representation. The dense layer then takes this integrated representation and performs the final classification, mapping the learned features to the corresponding character classes.

D. After defining the model architecture, there is a `call` method in the CustomCNN class. This method defines the forward pass of the model. This method defines the forward pass of the model, specifying the sequence of layers the input data (`inputs`) will pass through to produce an output. It sequentially applies the defined layers to the input and returns the final output of the dense layer.

Model Compilation

A. Defines the input shape for the model. The input image size is 64x64 pixels and a single channel (grayscale).
B. An instance of the CustomCNN model is created by calling CustomCNN(number_of_characters=len(label_encoder.classes_)). The `number_of_characters` argument is set to the number of classes in the dataset, which is obtained from `label_encoder.classes_`.
C. `model.compile` configures the model for training:
    a. Optimizer: `Adam` is used as the optimization algorithm.
    b. Loss Function: `sparse_categorical_crossentropy` is suitable for multi-class classification problems where the labels are integers.
    c. Metrics: The model's performance is evaluated using accuracy.

Model Training

A. early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True): The EarlyStopping callback is used to stop the training process early if the validation loss metric does not improve for certain number of epochs (patience). `monitor='val_loss'` : This tells the callback to monitor the validation loss during training. `patience=10`: Training will continue for 10 more epochs after the monitored metric (validation loss, in this case) has stopped improving. This is to ensure that stopping is not premature and that the model has genuinely stopped improving. `restore_best_weights=True`: Upon stopping,

the model weights from the epoch with the best value of the monitored metric (lowest validation loss) are restored. This means that even if the model's performance deteriorated in the epochs following the best one, it still ended up with the best performing model.

B. reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001): The ReduceLROnPlateau callback is used to reduce the learning rate when the validation loss metric plateaus. It reduces the learning rate by a factor (factor=0.2) when no improvement is seen for a certain number of epochs (patience = 5). It also has an option to set a minimum learning rate (min_lr=0.001), indicating the lower bound on the learning rate. No matter how many times the callback reduces the learning rate, it won't go below this value.

C. history_cnn = model.fit(...): The fit method is used to train a model on a given dataset. It takes several arguments:
    a. train_generator: The training data generator, which generates batches of training data.
    b. steps_per_epoch: The number of steps (batches) to take per epoch. In this case, it is calculated as the length of the training data divided by the batch size (len(X_train) // 32).
    c. epochs: Sets a high number for epochs (epochs = 1000), potentially allowing for extensive training. However, due to the EarlyStopping callback, training is likely to stop much earlier than this if no improvement is seen.
    d. validation_data: Uses the test_generator to provide validation data at the end of each epoch, allowing the model to be evaluated on a separate set of data not seen during training.
    e. callbacks: A list of callback functions to be called during training. In this case, it includes the early_stopping and reduce_lr callbacks.

Appendix IV

Model Architecture

Within the __init__ method, the model architecture is defined. The architecture consists of several layers:

A. Convolutional Layer 1: This layer applies 6 filters of size 5x5 to the input data. The activation function used is ReLU (Rectified Linear Unit), which introduces non-linearity to the model. The padding is set to 'same', which means the input and output dimensions of the layer will be the same. After the convolutional operation, a batch normalisation layer (BatchNormalization) is applied to normalise the outputs. Finally, a max pooling layer (MaxPooling2D) with a pool size of 2x2 is used to reduce the spatial dimensions of the output.

B. Convolutional Layer 2: Similar to the first layer, this layer applies 12 filters of size 3x3 to the input data. Again, ReLU activation, batch normalisation, and max pooling are applied.
C. Convolutional Layer 3: Increased convolutional layer. This layer applies 24 filters of size 3x3 to the input data. ReLU activation, batch normalisation, and max pooling are applied.
D. Fully Connected Layer: The output from the last convolutional layer is flattened into a 1-dimensional vector using the Flatten layer. This vector is then fed into a dense layer (Dense) with number_of_characters units. The activation function used in this layer is softmax, which produces a probability distribution over the different classes.

The call method is also defined in the CustomCNN_2 class. This method implements the forward pass of the model. It takes an input tensor `inputs` and applies the defined layers in sequence. The output of the fully connected layer is returned as the final output of the model.

After the class definition, the code sets the input_shape variable to (64, 64, 1), which represents the size of the input images expected by the model. The number 1 indicates that the images are grayscale, as they have only one channel.

Model Compilation

An instance of the CustomCNN_2 model is created, passing the number of classes (len(label_encoder.classes_)) as the number_of_characters parameter. This ensures that the final dense layer of the model has the correct number of units for classification.

The model is then compiled using the compile method. The optimizer is set to 'adam', which is a popular optimization algorithm for training neural networks. The loss function is set to 'sparse_categorical_crossentropy', which is commonly used for multi-class classification problems. The metric 'accuracy' is also specified to evaluate the performance of the model during training.

Appendix V

Model Architecture

A. Defines a function called `build_icr_model_with_gru` that takes two parameters: `input_shape` and `num_classes` using a combination of Convolutional Neural Network (CNN) and Gated Recurrent Unit (GRU) layers.
B. Feature Extraction Layers: These layers are dedicated to processing raw image data, extracting relevant features through convolution and dimensionality reduction via pooling.

a. Input Layer: Utilises the `Input` function, setting the shape of incoming images to 128x128 pixels with 3 colour channels.
b. Convolutional Layers:
    i. `conv2d_1`: Applies 32 filters of 3x3 size, uses ReLU activation, and maintains the original input size with same padding.
    ii. `conv2d_2`: Further deepens the feature map with 64 filters of the same size, again with ReLU activation.
    iii. `conv2d_50`: Continues the pattern recognition process, using 64 filters to enhance the network's understanding of the input.
c. Max Pooling Layers:
    i. `max_pooling2d_1`: Reduces spatial dimensions by half, simplifying the output from conv2d_1.
    ii. `max_pooling2d_2`: Applies the same dimensionality reduction to the output of conv2d_2.
    iii. `max_pooling2d_3`: Continues to condense spatial features from conv2d_3, streamlining data for the next phase.

C. Integration Layer: This layer's role is to reshape the multidimensional data from the feature extraction phase into a format suitable for sequential processing.
    a. `reshape_1`: Converts the pooled feature maps into a flattened form, preparing the data structure for the GRU layer. The Reshape layer is used to flatten the output by combining the width and channels dimensions into a single dimension.

D. Sequential Processing Layer: Defines the RNN layers using the GRU architecture to process the reshaped features sequentially, capturing temporal and spatial dependencies within the data.
    a. `gru`: The GRU layer is configured with 128 units and return_sequences=False, which means it will return the output of the last time step only.

E. Classification Layer: The final step in the network, this layer is responsible for classifying the processed features into predefined categories. This layer maps the output of the RNN layers to the number of classes in the classification task.
    a. `output`: Dense Layer which is a fully connected layer. The num_classes parameter is used to determine the number of units in this layer. The `activation` function used in this case is softmax, which produces a probability distribution over the classes.

F. Finally, the code creates an instance of the Model class from Keras, specifying the input and output layers. The model is then returned by the function.

Model Compilation

A. Defines the input shape and number of classes for the model. The input image size is 128x128 pixels and three channels (RGB).
B. An instance of the build_icr_model_with_gru model is created by calling build_icr_model_with_gru(input_shape, num_classes) and passing in two arguments: input_shape and num_classes.
C. `model.compile` configures the model for training:
    a. Optimizer: `Adam` is used as the optimization algorithm.
    b. Loss Function: `sparse_categorical_crossentropy` is suitable for multi-class classification problems where the labels are integers.
    c. Metrics: The model's performance is evaluated using accuracy.

Model Training

A. Create a ReduceLROnPlateau callback
    a. monitor: This parameter specifies the metric to monitor for reducing the learning rate, which means the callback will monitor the validation loss (val_loss). The learning rate will be reduced when the validation loss stops improving.
    b. factor: This parameter determines the factor by which the learning rate will be reduced. It is set to 0.1, which means the learning rate will be multiplied by 0.1 when the callback is triggered.
    c. patience: This parameter defines the number of epochs with no improvement after which the learning rate will be reduced. It is set to 5, which means if there is no improvement in the monitored metric for 5 consecutive epochs, the learning rate will be reduced.
    d. verbose: This parameter controls the verbosity of the callback. It is set to 1, which means the callback will print a message when it takes action, such as reducing the learning rate.
B. Set up Early Stopping
    a. monitor: This parameter specifies the metric to monitor during training. In this case, it is set to 'val_loss', which means the validation loss will be monitored. The validation loss is a metric that measures how well the model is performing on unseen data.
    b. patience: This parameter determines the number of epochs with no improvement in the monitored metric after which the training will be stopped. It is set to 5, which means that if the validation loss does not improve for 5 consecutive epochs, the training process will be stopped.
    c. restore_best_weights: This parameter determines restore model weights from the epoch with the best value of the monitored quantity. When restore_best_weights is set to True, the EarlyStopping callback will restore the model weights to the values they had at the epoch with the best validation

loss. This can be useful because it allows us to use the model with the best performance on unseen data, rather than the model at the end of training which may have started overfitting.

Overall, the EarlyStopping callback is a useful tool to prevent overfitting and save computational resources by stopping the training process early if the model's performance does not improve. By creating an instance of the ReduceLROnPlateau callback and customising its parameters, you can incorporate learning rate reduction into your model training process. This can help improve the model's performance by allowing it to adapt the learning rate based on the validation loss.

Appendix VI

Model Training

A. model.fit(): This is a method call on the model object.
B. train_generator: This is the first argument sent to the fit() function. It represents the training datasets created by the data generator, which provides the model's training data.
C. steps_per_epoch: This is the second argument passed to the fit() method. It determines the number of batches per epoch. len(X_train) // 32 is used to calculate the number of batches. len(X_train) represents the total number of training samples, and // is the floor division operator, which divides and rounds down to the nearest integer. The value 32 represents the batch size.
D. epochs=50: This is the third argument passed to the fit() method. It specifies the number of times the model will iterate over the entire training dataset. In this case, the model will go through the entire dataset 50 times.
E. validation_data: This is the fourth argument passed to the fit() method. It represents the data generator that will provide the validation data for the model. During training, the model's performance on the validation data is evaluated to monitor its progress and prevent overfitting.
F. callbacks: This is the fifth argument passed to the fit() method. It is a list of callback functions that will be called during training at specific points. Callbacks can be used to perform actions, in this case, early_stopping and reduce_lr are used as callbacks.
G. history_imageGen: This is the variable that will store the training history returned by the fit() method. The training history contains information about the model's performance during training, such as loss and accuracy metrics.

Appendix VII

Model Architecture

The `ICRHyperModel` class is defined using the class keyword. It inherits from the HyperModel class. The __init__ method is the constructor of the ICRHyperModel class. It takes two parameters: `input_shape` and `num_classes`. The constructor initialises these values as instance variables.

The build method is a required method in a hypermodel class. It is responsible for defining the architecture of the model. It takes a single parameter hp, which represents the hyperparameters that will be tuned during the hyperparameter search. Inside the build method, the model architecture is defined using the Keras functional API.

A. Input Layer:
    a. input_img = Input(shape=self.input_shape, name='image_input'): This defines the input layer of the model using the shape specified during class instantiation.
B. Convolutional Layers:
    a. A loop is used to create a variable number of convolutional layers. The number of convolutional layers is determined by the hyperparameter num_conv_layers, which is sampled from a range defined by min_value = 1 and max_value = 5. Within each iteration of the loop, the number of filters and kernel size for the convolutional layer are also sampled from their respective ranges.
    b. Each convolutional layer consists of:
        i. A convolutional process that utilises a variable quantity of filters, determined by the hyperparameter `conv_i_filters`. This parameter selects from a specified range with a minimum of 32 and a maximum of 256, incrementing in steps of 32. Additionally, the kernel size, governed by the hyperparameter `conv_i_kernel`, is chosen from a range with a lower bound of 3 and an upper limit of 7.
        ii. Rectified Linear Unit (ReLU) activation function.
        iii. Padding set to 'same' to preserve spatial dimensions.
        iv. Max-pooling operation with a 2x2 pool size to downsample the feature maps.
C. Reshape Layer:
    a. After the convolutional layers, the output is reshaped to prepare it for processing by recurrent layers.
    b. The output from the convolutional layers has a shape [batch_size, height, width, channels]. It is reshaped to have a new shape (-1, conv_shape[2] * conv_shape[3]) where conv_shape is the shape of the feature maps from the convolutional layers. This step combines the width and channel dimensions.

D. Recurrent Layers (GRU):
    a. The reshaped output is passed through one or more Gated Recurrent Unit (GRU) layers.
    b. The number of GRU units is a hyperparameter specified as gru_units from a specified range with a minimum of 64 and a maximum of 256, incrementing in steps of 64.
    c. These layers can capture sequential patterns in the flattened feature map.
E. Fully Connected Layer (Output Layer):
    a. The output of the GRU layer(s) is connected to a fully connected layer with num_classes units.
    b. A softmax activation function is applied to the output layer to convert the network's final output into class probabilities. This allows the model to predict the probability distribution over character classes.

The model is compiled with an optimizer, loss function, and metrics in the defined hypermodel class. The optimizer is defined using the Adam optimizer with a learning rate sampled from a set of predefined values. The loss function is set to 'sparse_categorical_crossentropy', which is commonly used for multi-class classification tasks. The accuracy metric is also specified.

Finally, the model is returned from the build method.

Tuning Process

Set the early stopping to monitor the val_accuracy with patience = 5. The search() method is used to perform the actual search for the optimal hyperparameters. It takes several arguments:

A. train_generator: The training data generator, an object that generates batches of training data for the model. The search() method will use this generator to train the model with different hyperparameter configurations.
B. steps_per_epoch: Specifies the number of steps (or batches) to take from the train_generator in each epoch. It is set to len(train_generator), which suggests that the code wants to iterate over the entire training dataset in each epoch.
C. epochs: Determines the number of times the search() method will iterate over the training data. It is set to 35, meaning that the search process will run for 35 epochs.
D. validation_data: The validation data generator. Similar to the train_generator, it is an object that generates batches of validation data for evaluating the model's performance during the search process.
E. validation_steps: Specifies the number of steps (or batches) to take from the validation_data in each epoch. It is set to len(test_generator), indicating that the code wants to iterate over the entire validation dataset in each epoch.

F. callbacks: A list of callback functions that will be called during the search process. In this case, the code is passing a single callback function called early_stopping. This callback is used to monitor the validation accuracy and stop the search process early if the validation accuracy does not improve for a certain number of epochs (specified by the patience parameter in the EarlyStopping callback).

After performing the tuner.search() method, use the `get_best_hyperparameters` method which returns the optimal hyperparameters (`best_hps`) as indicated by the objective and this approach may be used to reinstantiate the best (untrained) model discovered during the search phase on the tuner object to obtain the best set of hyperparameters discovered during the tuning process [32]. `num_trials=1` specifies that I want the best hyperparameters from the single best trial. The method returns a list of hyperparameter objects, and [0] is used to get the first (and in this case, the only) set of hyperparameters from the list.

And build the Model with the Best Hyperparameters by using the build method of the hypermodel instance to construct a new model instance using the best hyperparameters (`best_hps`) found by the tuner. Hypermodel is an instance of a class that defines the model architecture and how it should be compiled. It must have a build method that accepts a hyperparameters object and returns a compiled model. The resulting `best_model1` is a model instance that's ready to be trained, evaluated, or used for predictions.

Appendix VIII

Model Compilation

A. Defines the input shape and number of classes for the model. The input image size is 128x128 pixels and three channels (RGB).
B. An instance of the build_icr_model_with_regu model is created by calling build_icr_model_with_regu(input_shape, num_classes) and passing in two arguments: input_shape and num_classes.
C. `model.compile` configures the model for training:
    a. Optimizer: `Adam` is used as the optimization algorithm.
    b. Loss Function: `sparse_categorical_crossentropy` is suitable for multi-class classification problems where the labels are integers.
    c. Metrics: The model's performance is evaluated using accuracy.

Model Training

D. Define the early stopping and ReduceLROnPlateau callbacks, which monitor val_loss while training. The patients for early stopping and ReduceLROnPlateau are 10 epochs and 5 epochs, respectively.

E. Train the model by using model.fit() method and set the length of each epoch's steps to len(X_train) // 32, for a total of 50 epochs of training.

Appendix XI

Model Architecture

Defines a function `build_icr_model_with_lstm1`, `build_icr_model_with_lstm2`, `build_icr_model_with_lstm3` that builds CNN models with different convolutional neural network (CNN) layer numbers which is 3, 4 and 5 convolutional layers with an LSTM layer for image classification. And after the training, we can easily find out that the best accuracy among three models is the third one which got the highest convolutional layer numbers.

A. The function takes four arguments:
    a. `input_shape`: The shape of the input images, `(128, 128, 3)` for 128x128 RGB images.
    b. `num_classes`: The number of classes in the classification task.
    c. `dropout_rate` (optional): The dropout rate for the dropout layers, default is 0.5.
    d. `l2_rate` (optional): The L2 regularisation rate for the convolutional and dense layers, default is 0.001.
B. The CNN part of the model consists of the following layers:
    a. A 2D convolutional layer with 32 filters of size (3, 3), ReLU activation, 'same' padding, and L2 regularisation.
    b. A 2D max-pooling layer with pool size (2, 2).
    c. Sets of 2D convolutional and max-pooling layers with 64 filters each.
    d. A dropout layer with the specified `dropout_rate`.
C. The output of the CNN part is reshaped to a 2D tensor with shape `(batch_size, height * width * channels)` to be compatible with the LSTM input.
D. An LSTM layer with 128 units is added, and it returns only the final output (return_sequences=False). A dropout layer with the specified `dropout_rate` is added after the LSTM layer.
E. A fully connected (dense) layer with `num_classes` units and a softmax activation is added to produce the final output. This layer also has L2 regularisation applied.

Model Compilation

After defining the model architecture, creates an instance of the model using the `build_icr_model_with_lstm3` function, providing the input shape `(128, 128, 3)` and the number of classes `62`.
Compile the model with 'adam' optimizer, 'sparse_categorical_crossentropy' for loss function and metric 'accuracy' as well.

Model Training

While training the model, I discovered that the number of epochs affects the model's accuracy. So, I trained the model with 50 and 100 epochs individually, and it turns out that the model trained with 100 epochs has greater accuracy.

A. Define the early stopping and ReduceLROnPlateau callbacks, which monitor val_loss while training. The patients for early stopping and ReduceLROnPlateau are 10 epochs and 5 epochs, respectively.
B. Train the model by using model.fit() method and set the length of each epoch's steps to len(X_train) // 32, for a total of 100 epochs of training.

Appendix X

Model Architecture

A. Foundational Structure

At the core, the basic CNN model begins with an input layer designed to receive images. For the discussed model, the input is likely structured as 62x62 pixel RGB images, where each pixel contains three colour channels (Red, Green, Blue). This input format is standard, providing a balance between resolution for feature detection and computational efficiency.

B. Feature Extraction Layers

The first real layer of the model, a 2D convolutional layer (`conv2d_6`), is where the process of feature extraction begins. Equipped with 32 filters, this layer convolves the filters over the input image to produce a set of 32 feature maps. These filters are designed to detect simple features such as edges and colours at various locations across the image. The output dimension of this layer is 60x60x32, slightly reduced from the input due to the convolution operation, commonly achieved with a kernel size of 3x3.

Following the convolutional layer is a MaxPooling2D layer (`max_pooling2d_6`), which performs down-sampling by reducing the spatial dimensions of the feature maps. Specifically, it reduces the size of each feature map to 30x30 by applying a 2x2 pooling operation. This reduction not only diminishes computational requirements for subsequent layers but also introduces translational invariance, making the model more robust to the exact location of features in the image.

C. Integration and Classification Layers

Transitioning from feature extraction to classification, the model employs a Flatten layer (`flatten_2`). This layer reshapes the pooled feature maps from a 3D tensor into a 1D vector, preparing the data for processing by fully connected layers. The act of flattening is crucial as it bridges the gap between convolutional operations and the dense layers that follow.

The penultimate component of the model is a dense layer (`dense_4`) with 128 units. This fully connected layer integrates the flattened features through a combination of weighted sums and nonlinear activations, enabling the network to learn complex patterns and relationships among the features extracted by previous layers.

Culminating the model is another dense layer (`dense_5`), this time with 30 units, corresponding to the model's output. This layer is designed to output a final prediction, mapping the integrated features to a probability distribution across 30 classes. The number of units indicates the model is tailored for a classification task with 30 possible categories.

Through a sequential arrangement of convolutional, pooling, and dense layers, it encapsulates the essence of CNN architectures: feature extraction followed by classification. Each layer plays a critical role, from detecting simple features in the early stages to making complex categorical distinctions in the output.

Best Hyper Model Architecture Classification

A. Architectural Overview

The model starts with an input layer implicitly designed to handle images of a specific size, suggested by the dimensions of the output from the first convolutional layer. Although the exact input dimensions are not detailed, the first convolutional layer's output suggests that the model processes images slightly larger than 64x64 pixels, likely with three colour channels (RGB).

B. Deep Feature Extraction

The journey through the model begins with a Conv2D layer (`conv2d_7`), equipped with 96 filters. This significant number of filters in the first layer allows the model to detect a wide variety of features right from the start, from simple edges to more complex patterns. The output size is 64x64 with 96 feature maps, indicating a very slight or no reduction from the input size, likely using a small kernel size and either 'same' padding or a large initial input size.

Following this, the model employs a MaxPooling2D layer (`max_pooling2d_7`), halving the dimensions to 32x32 while retaining the 96 feature maps. This reduction emphasises the most prominent features detected by the preceding convolutional layer, improving computational efficiency and model generalizability.

The model continues to alternate between convolutional layers and max pooling layers, a strategy that progressively refines the feature maps. The second Conv2D layer (`conv2d_8`), with the same number of filters as the first but operating on the reduced dimensions, further elaborates on the features detected initially. This process of alternating convolution

and pooling continues, with each convolutional layer—conv2d_9, conv2d_10, and conv2d_11—followed by a max pooling layer, reducing the feature map size and adjusting the depth to capture different levels of abstraction within the image.

C. From Spatial Features to Classification

The model's final Conv2D layer outputs feature maps of size 4x4 with 96 channels, which are then passed through one last MaxPooling2D layer, resulting in a 2x2x96 output. This output is the distilled essence of the input image, transformed into a high-level, abstract representation that captures the most critical features for classification.

The Flatten layer (flatten_3) transitions the model from convolutional layers to fully connected layers by converting the 3D feature maps into a 1D vector. This vector, now a comprehensive feature representation of the input image, is ready to be processed by the dense layers for classification.

The dense layer (dense_6) with 384 units, matching the number of features from the flattened layer, allows for a full integration of these features. It serves as the backbone of the model's classification mechanism, where the learned abstract representations are interpreted to make predictions.

Finally, the model culminates with a dense output layer (dense_7) comprising 30 units, corresponding to the model's classification categories. This layer translates the integrated features into probabilities across 30 classes, making the final prediction.

This architecture is not just about depth but also about the thoughtful arrangement of layers to maximise feature extraction and efficiency. Through its intricate design, this model achieves a delicate balance between capturing a wide array of features and maintaining computational efficiency, making it a powerful tool for complex image classification tasks.