Automatic Number Plate Recognition using Machine Learning Techniques

Reuben Kurian

May 2024

Abstract

The rapid increase in transportation via vehicular means over the past 50 years has led to more incidents occurring on the road. In recent years, the swift expansion in artificial intelligence has brought about technologies such as automatic number plate recognition (ANPR). The main objective of this project is to build an ANPR system that can be accessed on a web app for users to access easily and in an uncomplicated manner. The system is comprised of three key components: number plate detection, character segmentation and character recognition in the listed order. The components will be connected in a pipeline-fashion and hosted on the web application. The web app is created on the Django framework, allowing for significant scalability. The number plate detection and character segmentation components are built using OpenCV and NumPy, and the character recognition component is a deep learning model utilising convolutional neural networks (CNNs) built using TensorFlow. The character recognition model's performance is compared against a character recognition model built upon the VGG16 architecture with transfer learning, examining the training accuracy and loss, validation accuracy and loss as well as precision, recall and F1-score. The web application is tested for functionality and usability. An evaluation of the results is made, comparing the results to the original project aims. A critical assessment is conducted, analysing the project's strengths, weaknesses, challenges faced and lessons learned.

I certify that all material in this dissertation which is not my own work has been identified.

Signed: Reuben Kurian

Contents

1 Introduction

Transportation constitutes a pivotal aspect of our daily lives, with a significant portion of road users consisting of car and lorry drivers, as well as cyclists. However, the high number of road users is associated with an increased frequency of road accidents. Annually, road accidents result in 1.35 million deaths and cause 20 to 50 million injuries. [3]. Of these accidents, Hit-and-run accidents continue to be among the most dangerous types of road accidents due to the lack of information about the vehicle responsible for the collision [11]. By capturing the license plate of the perpetrator before fleeing, a wealth of information can be acquired. Automatic number plate recognition (ANPR) can analyse input images containing license plates and return the identified license plate. This license plate information can then be forwarded to an API. For instance, the DVLA's VES API can accept an inputted license plate and provide details on the vehicle, including the vehicle's make, model, year of manufacture, colour, engine size, and various other information. Machine learning (ML) algorithms can be implemented to create models that will undertake different steps of the ANPR process. This paper focuses on implementing machine learning techniques to achieve automatic number plate recognition. Convolutional neural networks (CNNs) were utilised as part of the models' architecture. The models were trained on datasets containing license plates that were found readily available online. The primary objective of this project was to integrate machine learning techniques and web development, with the goal of producing ANPR software that is hosted on a web application that takes in an input image of a license plate and returns details on the vehicle. There are two major components to this project. The first being the ANPR software, which was created using Python, with the support of OpenCV and TensorFlow libraries. The second major component to this project is the web application. Incorporating a simple, clean and user-friendly user interface allowing for easy access to the ANPR software is paramount.
The aims of this project include:

- Creating a well-performing ANPR software using machine learning techniques
- Developing a web application that provides the user with an interface to operate the ANPR software.

The report is structured as follows:

- A literature review section analysing the key findings that are relevant to this project.
- A project specification section outlining the objectives, goals and requirements.
- A design section to define how my project will be approached, giving a basic conceptual plan to my project.
- A development section examining how each part of my project was implemented, any deviations from the original plan and why I made these changes.
- An evaluation section containing results obtained from testing and assessing these results against my main objectives.
- A critical assessment of my project, conducting a thorough analysis of the project's strengths, weaknesses, challenges and lessons learned.
- A conclusion, reflecting and summarising the entire project.

2 Literature Review and Project Specification

2.1 Literature Review

Automatic Number Plate Recognition (ANPR), alternatively referred to as license plate recognition, is a software application that employs computer vision and machine learning methodologies to discern number plates from images or videos. Its primary deployment occurs in car parking facilities, where a

mounted camera scans the front number plate of a vehicle upon entry [15]. An inductive loop detector initiates the ANPR system, prompting the extraction and decoding of the plate image. Subsequently, the number plate is either compared against a pre-existing database or stored for subsequent analysis [15]. The conventional ANPR system is delineated into four core components: vehicle image capture, number plate detection, character segmentation, and character recognition [18]. ANPR finds utility beyond parking management, extending to applications such as traffic safety enforcement and automated toll collection [18]. Optical Character Recognition (OCR) serves as the foundational technology enabling ANPR to interpret license plates [1].

Image preprocessing constitutes a pivotal stage [2] within the domain of image recognition. Various techniques are employed for this purpose, including mean normalization, standardization, and zero component analysis [17]. This preparatory step is integral to enhancing the quality of an image, thereby facilitating more effective analysis [14].

In the context of face image preprocessing, [10] employs a diverse array of methods. The preprocessing procedure is delineated into discrete stages [10], encompassing color normalization techniques such as intensity normalization, grey world normalization, comprehensive color image normalization, HSV hue, and BGI hue adjustments.

Furthermore, [10] employs statistical methodologies to ensure consistency in the brightness and intensity of images. These techniques encompass brightness variation, horizontal and vertical brightness adjustments, local brightness modulation, and mean brightness normalization to maintain uniform brightness and contrast across all face images.

Subsequently, convolutional methods are applied [10], encompassing adjustments for smoothness, blur, edge detection, contour enhancement, detail extraction, sharpening, and embossing. These convolution techniques serve to refine or diminish image characteristics, mitigate noise, and extract pertinent features [10].

Lastly, [10] adopts methodological combinations to leverage the benefits of multiple image processing techniques. These combinations include sequences such as "contour filtering followed by smoothing," "smoothing followed by contour filtering," "local brightness transformation followed by smoothing," "local brightness transformation followed by contour filtering," contour filtering integrated with local brightness transformation, "contour filtering followed by smoothing, augmented with local brightness transformation," and "smoothing followed by local brightness transformation, succeeded by contour filtering."

Optical Character Recognition, commonly referred to as OCR, represents a computer vision technology that employs an optical mechanism akin to human visual perception [16]. Integral to Automatic Number Plate Recognition (ANPR), OCR plays a vital role in deciphering number plates from inputted images and videos. The efficacy of OCR is closely tied to the quality of the input data, with factors such as higher resolution images/videos, optimal angle alignment of the number plate, favourable lighting conditions, and closer proximity to the image/video contributing to enhanced accuracy in results. Beyond ANPR, OCR finds application in various contexts, including the conversion of scanned paper documents into PDF files [16].

The construction of a computer system capable of autonomously learning and improving performance with experience is a topic of considerable significance, explored within the domain of machine learning [12]. This field not only addresses this fundamental question but also contributes extensively to the development of highly advanced computer software across numerous domains. Machine learning [5] represents a dynamic discipline within computational science, centered on the creation and refinement of algorithms designed to emulate human intelligence by assimilating knowledge from their environment. Situated within the broader realm of artificial intelligence (AI), machine learning [12] has emerged as the preferred approach for the creation of software in diverse fields such as computer vision, speech recognition, natural language processing, robotics, and various other domains.

Convolutional Neural Networks (CNNs) have demonstrated remarkable efficacy [19] across a spectrum of computer vision tasks. CNNs operate by receiving input in the form of a three-dimensional tensor. For instance, an image typically comprises H rows, W columns, and 3 channels, where H represents the image's height, W its width, and the 3 channels denote the presence of RGB color channels (red, green, blue). Subsequently, the input undergoes a sequence of computational steps. [19] encapsulates the

successive progression of a Convolutional Neural Network (CNN) through its layers during a forward pass. The notation x1 represents the input, commonly comprising a three-dimensional tensor image. The parameters governing the output of the initial layer's computation are collectively denoted as a tensor w1. Following this, x2 denotes the output of the first layer, serving as input for subsequent layer computations. This iterative process persists until all layers within the CNN have completed their operations, culminating in xL. An additional layer is incorporated for the purpose of backward error propagation, a method employed to refine parameter values for optimal learning within the CNN.

2.2 Project Specification

In the scope of this project, the primary objective was to devise a machine learning model intended for the purpose of automatic number plate recognition (ANPR), hosted within a web application interface, offering users effortless access to the ANPR system. This model was designed to accurately detect number plates with a high degree of accuracy, while concurrently extracting pertinent details regarding the corresponding vehicles, utilising the DVLA VES API. Moreover, a key aspect of this project involved the comparative assessment of the performance exhibited by the character recognition model against a model created via leveraging transfer learning on a pre-trained model such as VGG16, a renowned image classification model.
The data required for this project was acquired from Roboflow and Kaggle. The ANPR software is comprised of three distinct components, namely the initial Python software tasked with extracting the number plate region from an image (number plate detection), Python software which is then ran on the detected license plate to detect character regions (character segmentation) and a deep learning model, all connected in a pipeline. The deep learning model will perform character recognition, which will read and determine what each character is within each bounding box.
Concerning the project's design, the implementation of number plate detection is constructed utilising Python in conjunction with OpenCV, a Python-based computer vision library. Additionally, supplementary libraries such as NumPy are employed in the process. Numerous preprocessing procedures were employed to extract the number plate region, such as applying masks, bilateral filtering, edge detection and adaptive thresholding. In the development of the ANPR software, the character recognition component was constructed utilising Python in conjunction with TensorFlow, an open-source machine learning library. TensorFlow facilitated the creation of the model, employing various optimisation strategies such as adjusting learning rates, epochs, model architecture, and integrating dropout techniques to mitigate overfitting. Following the construction of this model, rigorous testing procedures were conducted to assess their performance. Performance evaluation metrics including precision, recall, and F1-score were employed to gauge the effectiveness of the models. These components were integrated in a sequential manner, forming a cohesive pipeline to enable the operation of the ANPR software.
In addition to the ANPR software, the development of the web application was undertaken utilising Python and Django. Subsequently, the web application will be deployed on Google Cloud's App Engine, facilitating public access for users to utilise the application.

3 Design

The overall design of the code can be split into four distinct sections. These consist of number plate detection, character detection, character recognition and the web app hosting the code. The first three sections operate sequentially in a pipeline fashion to facilitate the creation of the ANPR software. Due to this pipeline fashion, the number plate detection was developed first.
When considering the ANPR software at a high level, it operates by inputting an image containing a number plate and producing the corresponding number plate information. Upon closer examination, the software accomplishes this task in a significantly more intricate manner.

3.1 Number Plate Detection

The number plate detection is in the form of a Python function, and it will take in the image path of an image and return a cropped image where the license plate resides in. Initially, there was a consideration to develop an object detection model to identify the license plate regions. However, due to time constraints and achieving comparable results with the current implementation, the decision was made to forgo the use of the object detection model.
The first part of the function involves preprocessing the image so that it is ready for number plate detection. Firstly, the image is converted from the RGB to the HSV colour space. This procedure is crucial for the next few steps, as images in HSV format provide enhanced colour segmentation, greater resilience against bright areas such as camera flashes and more effective thresholding. Subsequently, the function defines the lower and upper bounds for the grey, yellow, and white colours. Masks are then created for these ranges and combined to obtain the regions of interest using the HSV image, as number plates are most commonly found as white, yellow or grey. This mask is then applied to the original image. The new masked image is then converted to greyscale, Images converted to greyscale allow for more efficient processing. Noise reduction is also employed in the form of bilateral filtering, smoothing the image while retaining the sharpness of the edges. Edge detection is performed to identify all possible edges within the image. Adaptive thresholding helps to further segment the regions of interest. Following this, the function identifies contours within the thresholded image, attempting to locate the license plate within the obtained contours. If the license plate is not found, the function recalculates a new set of contours on the image with edge detection applied. Once a suitable contour is found, the function returns the cropped image containing the license plate.

3.2 Character Segmentation

The initial plan for character segmentation involved the development and training of a model to identify regions containing characters on the license plate. However, this approach was ultimately abandoned in favor of utilising image processing techniques similar to those employed in the number plate detection section. Initially, the input image is transformed into grayscale, effectively removing colour information. This step facilitates subsequent processing stages. Gaussian blurring is then applied to the grayscale image to reduce noise and enhance stability for further processing. After this, adaptive thresholding is performed to binarize the image, simplifying the segmentation of letters from the background of the license plate. Connected component analysis (CCA) is then carried out, which when used in conjunction with filtering the components by their size, helps to identify unwanted blobs on the license plate that are not characters. Contours are then found on the image, which are then sorted and returned. The output of this function will then be fed forward to the character recognition section.

3.3 Character Recognition

3.3.1 Dataset

Before proceeding with the model training, it was necessary to obtain a dataset. A dataset containing license plate images with annotations, including information such as the filename, image sizes, bounding boxes, and classes, was sourced from Roboflow. The dataset is structured in such a way that it is already divided into training, testing, and validation folders. Subsequently, the dataset was prepared to be used by the created model. Both the annotations and images are loaded using proprietary functions created, employing various libraries to streamline the process. However, due to the dataset being small, I also decided to combine this dataset with a different dataset I found on Kaggle. Having a composition of both datasets allows the model to adapt to a larger variety of images and ensures consistency, at the cost of increase data preparation overhead.

3.3.2 Model Configuration

Convolutional neural networks (CNNs) are utilised as part of my model configuration due to its ability in effective feature extraction. The model is compiled using a lower learning rate than the default

learning rate, and then trained on the training images and labels, harnessing the validation images and labels as validation data. During training, the training accuracy, training loss, validation accuracy and validation loss are all displayed in real-time as the model is training, with the values of these metrics finalised at the end of each epoch. The model was saved to a separate folder so that the model could be used for testing, evaluation and in the final ANPR pipeline if it performed well. The model was assessed using the testing dataset, which was preprocessed to ensure that the labels are correctly matched to the images. Images from the testing dataset containing the predicted characters were displayed to demonstrate the effectiveness of the model. Additionally, a plot illustrating the training and validation loss was generated.

3.4 Web Application

The fourth section of the code encompasses a web application built using Django, an open-source web framework available in Python, and publicly hosted using Google Cloud's App Engine. The folder for the web application is named as "anpr-project/ANPR", containing ANPR and myapp subfolders. The settings for the web application are located within the settings.py file of the ANPR subfolder. The views.py file of the myapp subfolder contains Python functions responsible for rendering various HTML and CSS files. The urls.py file within the myapp subfolder establishes the path and calls the function from views.py. Additionally, the urls.py file within the ANPR subfolder includes all URLs from the myapp/urls.py file. A models.py file is included within the myapp folder for implementing database models; however, since databases were not utilised, this file remains empty. Files such as asgi.py, wsgi.py and __init__.py within the ANPR subfolder and __init__.py, apps.py and admin.py are also left in their default forms.
A few files are required for deploying the app onto the App Engine server. Within the parent ANPR folder, a requirements.txt and app.yaml file are created to enable deployment. The requirements.txt file contains all the necessary Python packages required to run the application and the app.yaml file is used to configure the application.
The ANPR software is hosted on this application, within the /home path. Users are directed to the homepage (/home) by default when entering the website. Upon accessing the homepage, users are presented with a clean and simple user interface. An "Upload Image" box is displayed, which prompts the user to upload an image when clicked. The image is then fed into the ANPR software pipeline which will then return a number plate.

3.5 Testing and Evaluation

Model testing will verify the performance of the character recognition model to ensure it performs to a high standard, while system testing will validate the proper integration and functioning of the number plate detection, character segmentation and the character recognition model in conjunction with the web application system. The models will be evaluated on test data, measuring the loss and accuracy. The model will also be manually tested on the test data, taking the generated predictions and comparing them against the true values. A classification report will be produced, giving a wide range of information such as precision, recall, F1-score and support.

4 Development

Development is the process of applying the aforementioned design theories in partnership with the information obtained from literature review into practice and bringing the project specification into fruition. This section will focus on how the ANPR software was developed, explaining what decisions were made and why I made these choices.

4.1 Number Plate Detection

The number plate detection algorithm was developed in Python using OpenCV as well as NumPy. I chose to use Python as it is a simple coding language to use, plus with the wide range of libraries for

computer vision and numerical computing, namely OpenCV and NumPy, it enables for a simple but effective development process.
The process began by identifying the start and end points of the algorithm, which were an image containing a license plate and the output being just the number plate region. The algorithm can be split into two parts, which can be labelled as contour identification and contour selection. The first part obtains the contours that could potentially represent the license plate in the image, and the second part loops through the obtained contours and based on certain criteria, will pick the best candidate. The first part begins by reading an image into the function by using the imread function offered by cv2 (the import name for the OpenCV library). The output is a NumPy array representing the image. The image is then resized using cv2.resize() to standardise the inputs and improves efficiency, especially if the inputted images are large. The NumPy array is then fed into the cvtColor function which takes in two parameters; the image and the colour format to be converted into. In my implementation, the image is converted into hsv format because of its robustness to lighting conditions as well as more effective thresholding. Grey, yellow and white colour ranges were defined in NumPy array format, and masks were created for each range. The masks were then combined using cv2.bitwise_or() and applied to the original image using cv2.bitwise_and(). The applied mask is utilised to identify potential number plate regions. These colours were chosen as they are the most common colours for number plates. The image is then converted to grayscale to simplify processing by removing the colour regions. Subsequently, bilateral filtering is employed using OpenCV's bilateralFilter() method to smooth the image while preserving edges. Adaptive thresholding is implemented on the filtered image to binarize the image. Pixel values are assigned either 0 or 255 based on its intensity. This helps highlight regions of interest such as the license plate. The potential contours are then found using the cv2.findContours() function, which will take the thresholded image and identify contour regions within the image. The contours are then filtered from the original list using the grab_contours() function from the imutils library, returning contours that are compatible with OpenCV, preventing errors. The points are then sorted using their contour area as the key, and sorted in reverse order to place the contours in descending order of area, resulting in the larger contours. such as potential license plate regions. at the front of the list.
The second part of the algorithm begins by initialising some variables. The minimum and maximum allowed aspect ratios for the contours are defined. A max time limit for the upcoming while loop is set at 5, and a start time is initialised by calling time.time() from the built-in time library. I set this so that the upcoming while loop would not be stuck in an infinite loop while looking for a location. A while loop is started, with the break condition being the time limit reaching five seconds. A for loop is placed within the while loop, where the for loop will iterate over every contour in the list of contours. Each contour will be approximated by a polygonal curve using the cv2.approxPolyDP() function, resulting in a shape around the contour region. If the length of the curve is equal to 4, a bounding rectangle will be formed from the curve using OpenCV's boundingRect() function, taking in the curve as a parameter. The output will be mapped to four points, and if the width and height are not equal to zero, or if the aspect ratio (calculated by dividing the width by the height) falls within the ideal range, the number plate region will be extracted from the original image using the four points and returned. If the approximated polygonal curve is not equal to four, the contours will be calculated again, this time using OpenCV's Canny edge detection algorithm instead of using adaptive thresholding. The grab_contours function from imutils is utilised again, and the contours are sorted before entering the loop again. Changing from adaptive thresholding to edge detection and vice versa could produce better results and makes the algorithm more robust by adapting to images where adaptive thresholding is the better algorithm or where edge detection is the better algorithm.
Error handling was also implemented into the algorithm to handle cases where the image path is invalid or if the image does not exist. Before the image is read in, the function checks if the image is None. If it is None, a print statement is executed and the function returns None.

4.2 Character Segmentation

The character segmentation algorithm also utilises the OpenCV and NumPy libraries, and developed on Python. The simplicity offered by Python and the libraries on offer makes Python a more than

viable option for character segmentation.
The algorithm begins by converting the image to greyscale, using the cv2.cvtColor() function. Gaussian blurring is applied to the greyscaled image using cv2.GaussianBlur(). Gaussian blurring reduces noise on the image and smoothens edges, producing a clearer image for character segmentation. Adaptive thresholding is then applied on the blurred image, producing an image which would contain only black or white pixels. Connected component analysis (CCA) is then carried out on the image. Connected component analysis is the grouping of pixels together in a binary image that belong to the same object. In layman's terms, it isolates the objects found in the thresholded image. A blank mask is created and initialised using the np.zeros() function, taking in the thresholded image's shape as the parameter. The image pixels are calculated, and based on the image pixels, an upper bound and lower bound for the objects are defined. This is to ensure that objects that are excessively small or excessively large are not mistaken for as a character. A for loop is initiated, iterating over every object in the detected objects found from the connected component analysis step. A labelMask is initialised using np.zeros() with the thresholded image's shape as the parameter, and then set to the label's values, resulting in a mask representing the objects on the image in NumPy array format. The non-zero values in the mask are counted using cv2.countNonZero(), and if this number falls within the upper and lower bounds, the labelMask is added to the blank mask. This is repeated until the for loop has iterated through every label. Following this, contours are found using cv2.findContours(), taking the mask as a parameter. The bounding boxes are then found by creating a list and applying cv2.boundingRect() for every contour in the previously obtained list of contours. The boundingRect() function computes bounding boxes around each contour, and returns the coordinates of the smallest bounding box encompassing each contour. The bounding boxes are then sorted based on a key. The functools library offers a cmp_to_key() function which takes in a compare() function as a parameter. The cmp_to_key() function is passed into the sorted() function as a key, sorting the bounding boxes vertically and if in the same vertical plane, then sorted horizontally. The sorting is paramount as it ensures the bounding boxes are ordered as they should on a license plate.
The compare() function I had created takes in two parameters, where in my instance would be two bounding boxes, examines the absolute difference in y-coordinates of the bounding boxes, and if it is greater than 10, return the difference, else return the difference in x-coordinates of the bounding boxes.

4.3 Character Recognition

4.3.1 Dataset

Before developing the character recognition section, since the plan was to create a character recognition model, it was mandatory to find a dataset. I found a dataset on Roboflow which contained over 500 training images, 200 validation images and 200 testing images.
The Roboflow dataset originally contained license plate images and the annotations contained information such as the filename, width and height of the image, the character the bounding box is surrounding on the number plate, and the four bounding box coordinates. Each row represented an individual bounding box. I needed characters for character detection and not the whole license plate, so I made Python code that would read the annotations, crop the characters out of each image in the dataset and save these cropped images into a new folder, and each image further being classed into different subfolders based on what character is in the image. Essentially, I converted the dataset from one that was meant for character segmentation, containing license plates, into a dataset that can be used for character detection. I initially trained the model on this dataset and swiftly realised a larger dataset was needed due to the model overfitting on the training data and poor accuracy with unseen data. I found a dataset on Kaggle with 24000 images in total and merged this dataset with the Roboflow dataset. The Roboflow dataset had already contained annotations in the form of a csv file, which I had written code to extract these annotations and construct the dataset. The Kaggle dataset however, contained annotations in various XML files, each file representing an image. I wrote some Python code to convert these XML annotations into a csv file and parse in the dataset in this manner. The conversion of the Kaggle dataset annotations into csv format provided more simplicity

as all the annotations were in the same format, leading to a simplified process of loading the dataset into Python.
The final dataset was structured as such:

- Three separate folders, where each folder was for training, validation or testing. Since the dataset was structured like this, it enabled for a more streamlined workflow and prevented data leakage.
- Each folder was further split into its respective labels, from 0-9 and A-Z, suggesting there are 36 subfolders in each folder.
- Each subfolder in the training folder contains over 500 images per class, whereas each subfolder in the validation folder contains over 180 images per class. The testing subfolder contains roughly 30 images per class.

After the combination of the two datasets, the total number of images amounted to more than 25000 images. Due to the model architecture I used and the lack of a GPU during training, I had to reduce the size of the dataset to around 10000. I did this by creating code that would randomly pick 210 images per class from the original training folder for the new training folder and 45 images per class from the original testing and validation folders for the new testing and validation folders. The reduction of the dataset provided a fair trade between model accuracy plus robustness and reasonable training times.
The dataset is loaded dynamically into Python using the ImageDataGenerator() function from TensorFlow. Within the function call, I can define parameters such as the rescale ratio (allowing for normalisation), and preprocessing techniques such as rotation. I also implemented a custom function where if the image is not able to be read, the exception is caught and an error message is printed. Catching the images that cannot be read or processed prevents the model from crashing during training and testing. A flow_from_directory() function call is made, where the location of the dataset is defined, as well as the batch size, image resize dimensions and the class mode. I chose to use this method of loading in the dataset instead of manually loading the images and mapping the labels to them as it is more computationally efficient using the ImageDataGenerator() since the flow_from_directory() function call loads the dataset on-the-fly.

4.3.2 Model Implementation

TensorFlow was the library of choice to use to create my character recognition model. I chose TensorFlow as it offers a high degree of flexibility and scalability, various levels of abstraction, performance adaptability and a large active community online, which was useful for troubleshooting and gathering intel during the development process. TensorFlow offers the Keras API, which can be utilised to develop deep learning neural network architectures. I used the Keras API and created a Sequential model, and the model is structured as follows: a convolutional layer is used as the input layer, with 32 filters, (3,3) kernel size, ReLU (rectified linear unit) as the activation function, and the input shape specified as (64,64,3). The next layer is a MaxPooling2D layer, where the pool size is (2,2). Another convolutional layer is added with the same parameters, this time with 64 filters, followed by a MaxPooling2D() layer of the same pool size as the previous pooling layer. A final convolutional layer is added with the same parameters but with 128 filters, followed by another MaxPooling2D() layer of the same pool size. A Flatten() layer is then added, followed by a Dense() layer with 256 neurons and the ReLU activation function, proceeded by a Dropout() layer with a dropout rate of 0.5. Another dense layer is added with 128 neurons with the ReLU activation function, followed by a dropout layer with a 0.5 dropout rate again. An extra Dense() layer is added, with 64 neurons and a ReLU activation function. The final layer of the model is a Dense() layer, with 36 neurons (representing the number of output neurons required for the model), with a 'softmax' activation function.

Figure 2 illustrates the aforementioned model architecture. The purpose of the convolutional layers in my model is for feature extraction [4]. I chose to only include three layers as once I started to add more convolutional layers, I was noticing significantly longer training times for diminishing returns. I found three layers to be enough for character detection as the images that will be fed into the model

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(36, activation='softmax')
])
```

Figure 1: My custom model architecture for character recognition.

will just be the character on a plain background, as opposed to other image classification tasks where more feature extraction is necessary to get more accurate results.
I also added the dropout layers as without them, training times were longer and the model also started to overfit on the training data. I compiled the model using the Adam optimiser, with the learning rate set at 0.0001, the loss function to be sparse categorical crossentropy and the training metrics to measure the accuracy. The model was then trained on the training data, using the validation dataset to validate the model and reduce overfitting. I trained the model over 20 epochs and created a ModelCheckpoint callback to save the model's best weights to a folder called best_tr_model1/best_model.h5. I found the HDF5 file format to be best to saving the model's weights, as I was getting issues with saving the weights in KERAS file format. I also implemented an EarlyStopping callback, which would stop the model's training when the validation loss has not improved over 3 consecutive epochs. A ReduceLROnPlateau callback was also employed, modifying the learning rate if no improvement has been made over a set duration. My ReduceLROnPlateau callback monitors the validation loss and divides the learning rate by 10 if the validation loss has not improved over four consecutive epochs. A minimum learning rate has been set at $1e{-}7$ to prevent the learning rate from being too low. The model.fit() function call was saved to the history variable, which was save the training loss and accuracy as well as validation loss and accuracy. These metrics were used later to visualise the model's training results.
I had also utilised a pre-trained model with transfer learning to fine-tune the model to my dataset to compare how a pre-trained model with a more complex architecture and weights that had been obtained from being trained on a larger dataset over more epochs would compare against a model that I had built from scratch using solely my own dataset.
[13] conducted a study where they compared four different neural network architectures and their performance on COVID-19 diagnosis using x-ray images. Of the four model architectures, VGG16 performed the best overall, scoring the best out of the four architectures in accuracy, specificity, false positive rate, positive predicted value and F1-score. I chose the VGG16 model as my model of choice, due to its high performance on image classification and recognition tasks.

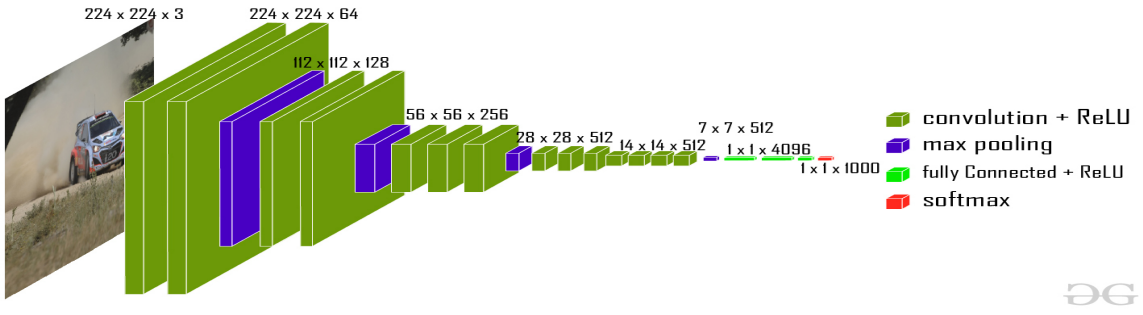

Figure 2: VGG16 model architecture.

Figure 2 [7] displays the architecture of the VGG16 model. As hinted by the name, the model has 16 layers, illustrated in Figure 2. The model contains 13 convolutional layers and 3 fully connected layers. Each convolutional layer is followed by a max-pooling layer, with the depth progressively increasing [7]. This enables the model to make robust and accurate predictions.

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(64, 64, 3))

base_model.trainable = False

model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(36, activation='softmax')
])
```

Figure 3: My implementation of the VGG-16 model using transfer learning.

Figure 3 portrays how I implemented the VGG-16 model with the use of transfer learning. I utilised the VGG-16 architecture by saving the model to a variable called base_model, and specifying the weights parameter to the 'ImageNet' dataset. The VGG-16 model architecture has been trained on the ImageNet dataset, which is a dataset consisting of over 14 million images [7] belonging to 1000 classses. The model has achieved an accuracy of 92.7%. Harnessing the weights of the model that was trained on this dataset provides us with transfer learning capabilities, which include but not limited to: improved feature extraction, improved performance, reduced overfitting, faster convergence and better generalisation. The include_top parameter has been set to 'False'. By defining the parameter to be false, the top layers (fully connected layers) are not loaded with the rest of the model. This is useful for my task for many reasons. Including the trained convolutional layers for feature extraction but excluding the dense layers allows for more tailored fine-tuning to my task of character recognition, considering that the dense layers are responsible for classification. My images also have different dimensions to the model. The ImageNet dataset contained images in the shape (224,224,3). My images vary, but are all resized to the shape (64,64,3). The model reflects this, as displayed in Figure 3. I ultimately chose my own convolutional neural network model shown in Figure 1 over other options such as support vector machines and random forests. [9] demonstrates that CNNs achieve higher accuracy than SVMs in image classification tasks.
I had trained the models on Google Colab as I do not have the necessary computational requirements to train the models comfortably and efficiently on my own device. Colab offers a CPU runtime of up to 12 hours, whereas the runtimes with GPUs available are often limited. Despite this, I would try to train the models on a GPU whenever possible, as it led to much faster training times.

4.4 Web Application

For the web application, I decided to use the Django framework. Django was released in July 2005 as an open source project, and currently is a leader in not only Python web frameworks, but in the web frameworks field in general [6]. Before implementation, I was contemplating between Flask and Django, but Django swiftly became an obvious decision for multiple reasons. Flask is more suited for small scale projects, is less scalable and requires more work to maintain and manage than Django [8]. Django is also more suitable for deployment if I planned on making the web application available publicly.
The Django folder resides within my project folder, and is called ANPR. Within the folder, there is a subfolder called ANPR, which is the master folder, containing the settings.py, the asgi.py, the urls.py the wsgi.py files. The settings.py file is the heart of the configuration, controlling the installed apps' settings, middleware settings, database connections and many more. The asgi.py file is used to support asynchronous features, and the wsgi.py file is utilised to allow web servers to interact with the

Django application on synchronous servers. The urls.py file in the ANPR subfolder is configured to contain every application's urls. In my instance, I included only myapp's urls for the reason that I am only developing one web application. The myapp subfolder contains the other group of files needed to run the application, such as HTML files, its own urls.py file, a views.py and a main_1.py file that contains the ANPR software. A subfolder called best_tr_model1 contains the character segmentation's model's weights. The HTML files are used to construct the web interface, the main_1.py file contains the functions necessary to run the ANPR software, the views.py file includes the functions that would be executed, and the urls.py file contains all the necessary paths to enter based on what functions within views.py are executed.
The server is started by entering the Django directory via terminal and entering the command 'python manage.py runserver', which will run the server on 'http://127.0.0.1:8000.
The user interface consists of the homepage, where the user is prompted with a greeting and a 'Go to ANPR' button, which will take them to the /run-pipeline path, where they will be greeted with a message and an 'Upload Image' button. Upon clicking the 'Upload Image' button, the user is prompted to upload an image from their device. proceeding this, the uploaded image will be fed into the ANPR software, starting with the number plate detection algorithm to extract the license plate region in the image, followed by the character segmentation stage, where each character is individually segmented and cropped, concluding with the character detection model, where each cropped image of the letters within the license plate is passed into the model, and the model determines what character is present within each cropped image. The results are formed in order, and returned to the user. The user is then prompted if it is accurate, if they would like to find out more information about the vehicle. If they click yes, the number plate is then sent within a POST request to the DVLA VES API, where a JSON file containing all the vehicle details are returned. The JSON file will be formatted in such a way that the information presented back to the user is user-friendly. All pages have the home button where the user can easily navigate back to the home screen.
The templates folder contains HTML files as previous mentioned. The base.html file is written in HTML and Jinja2. Jinja2 is a templating engine that can be useful for creating HTML templates, allowing for greater code reusability. Bootstrap CSS templates hosted via content delivery network (CDN) were used to simplify the process, in addition to Bootstrap JavaScript functionality to handle the functionality of the The home.html file is given a foundation by simply writing code that extends the base.html file. More functionality can be added on top of the extended base. In my instance, I added code to handle the ANPR software on the front-end, by providing buttons and outputs the results in an orderly fashion. I had ultimately decided not to host the website on Google Cloud's App Engine due to financial reasons with hosting the website on a public server. However, the software is ready to be deployed and easily scalable as a consequence of developing the web app on the Django framework.

5 Results

5.1 Number Plate Detection and Character Segmentation

Due to these components not utilising a model, testing was done manually. Manually running the algorithms on their own against various images helped to gauge what the accuracy is.
As demonstrated in figure 4, the components are working as they should. During testing, it was observed that occasionally the region identified as the 'detected' license plate did not correspond to an actual license plate but rather represented another object or area within the image. And with the character segmentation algorithm, the bounding boxes may fail to capture certain letters or capture parts of the license plate that are not relevant to the license plate itself. The algorithm demonstrates good performance when presented with images exhibiting clear visibility of the number plate, characterised by good lighting conditions and the image not being as high quality. I found images below 1000 pixels in either height or width to perform best on the algorithm. This is due to the fact that a higher resolution image, even when resized to a smaller size, will retain more detail than an image already at the resized size, hence leading to more contours detected and therefore a much more difficult process in detecting the correct regions.

Figure 4: Visual depiction of how the license plate detection and character segmentation takes place.

5.2 Character Recognition

Test images and test labels were loaded into Python manually, as I was having issues using the ImageDataGenerator function that I was previously using for training and validation data. The main issue I had to deal with was not being able to map the predictions to its decoded values, however manually loading the dataset resolves this as I had more control over how the testing dataset was formed. I initialised two empty lists, labelled test_images and test_labels. A for loop was executed, where it would loop through every subfolder in my testing dataset, and for every subfolder, load in the image, resize it to 64x64, convert it to a NumPy array and normalise it by dividing it by 255, before appending the image and the label to its respective lists. X_test and y_test variables were created by taking the test_images and test_labels lists and storing them as NumPy arrays respectively. The scikit-learn library offers a LabelEncoder() function, which I used to transform the labels. It encodes the classes from 0-9 and A-Z as numbers, so that the model can predict the values properly as letters cannot be passed in. The model is predicted on these encoded values, and for testing purposes, I applied the inverse_transform function offered by scikit-learn to convert the predictions into predicted labels.
For the character segmentation section, results were obtained by building a classification report on the test. The scikit-learn library offers a classification_report function, which takes in the test_labels and the predicted labels and returns a summary of the precision, recall, F1-score and support for each class.

5.2.1 Custom CNN Character Recognition Model Results

.
The results of the classification report performed on the custom character recognition model are shown in figure 5. It is evident from the results that the model is performing very well on all classes, except for classes '0' and 'O'. Class 0's precision, recall and F1-score are 0.68, 0.59 and 0.63 respectively when the macro average of these metrics are all 0.97. Class O's precision, recall and F1-score are 0.59, 0.68 and 0.63 respectively. Both classes having similar results and also being much worse results than the average displays the difficulty the model had discerning between the two characters compared to the other characters. Class C also has a slightly worse F1-score and precision than the average, although not as drastic as class 0 and class O. Excluding this, the model is performing well on the testing dataset on the other classes, meaning the overall accuracy is high at 0.97, as well as macro and weighted averages are high, at 0.97 for all metrics for both averages.
Results were also obtained by extracting the training accuracy, training loss, validation accuracy and validation loss from the history variable stored from training the model, and plotted on two separate graphs. The graphs were plotted using matplotlib's pyplot, plotting the accuracy and loss (on separate graphs) on the y-axis, and the epochs on the x-axis.

Figure 6 illustrates the variation of training and validation accuracy over epochs as the model progresses. The training accuracy is at 5% accuracy at the start and increases at a fast and almost-linear rate up to the third epoch towards 60% accuracy. The accuracy increases at a slower rate, finishing at 82% accuracy by the 20th epoch.

Class	Metrics			Support
	Precision	Recall	F1-score	
0	0.68	0.59	0.63	32
1	0.92	1.00	0.96	34
2	1.00	1.00	1.00	32
3	1.00	0.97	0.99	35
4	1.00	0.97	0.99	35
5	1.00	0.97	0.99	35
6	1.00	0.97	0.98	33
7	1.00	1.00	1.00	34
8	1.00	1.00	1.00	36
9	1.00	1.00	1.00	39
A	0.97	1.00	0.99	37
B	1.00	1.00	1.00	36
C	0.80	1.00	0.89	35
D	1.00	1.00	1.00	33
E	0.97	1.00	0.98	32
F	1.00	0.97	0.98	31
G	1.00	0.70	0.82	33
H	1.00	1.00	1.00	35
I	1.00	1.00	1.00	28
J	1.00	0.96	0.98	28
K	1.00	1.00	1.00	31
L	1.00	1.00	1.00	33
M	0.94	1.00	0.97	34
N	1.00	1.00	1.00	34
O	0.59	0.68	0.63	28
P	1.00	1.00	1.00	35
Q	1.00	1.00	1.00	28
R	0.97	1.00	0.99	33
S	1.00	1.00	1.00	37
T	1.00	1.00	1.00	31
U	1.00	1.00	1.00	37
V	1.00	1.00	1.00	32
W	1.00	1.00	1.00	28
X	1.00	0.97	0.99	37
Y	1.00	1.00	1.00	33
Z	1.00	1.00	1.00	39
Overall Metrics				
Accuracy		0.97		1203
Macro Avg	0.97	0.97	0.97	1203
Weighted Avg	0.97	0.97	0.97	1203

Figure 5: Classification report of my custom character recognition model.

From analysing the graph, it is blatant that the validation accuracy begins higher, at 15%, and increases at a much more rapid and linear rate up to 65-70% by the first epoch. From the first epoch to the second, the rate decreases, going up to 85-90% accuracy. The accuracy increases to 90-92%, before tapering off at 96% at the eighth epoch, where it fluctuates between 95-97% until the 20th epoch, finishing at 96%.
Training and validation loss are plotted against the epochs, illustrated in figure 7. The training loss starts at 3.6, and decreases almost linearly to 2.1 by the second epoch. By the fourth epoch, the model's training loss is at 1.4. From the fourth epoch until the final epoch, the loss decreases at a much slower rate, ending training with a loss of 0.5.
The validation loss evidently begins at a lower point, starting at a loss of 3.4 and decreases linearly to the first epoch at a faster rate than the training loss, having a loss of 1.7. The model's validation loss decreases at a less steep rate to the second epoch, having a loss of 1.0. By the third epoch, the validation loss is at 0.6, and by the fourth it reaches 0.4. The rate of the validation loss decreasing tapers off, and reaches a loss of 0.2 by the 20th epoch.

5.2.2 Pre-trained Character Recognition Model Results

.
Figure 8 portrays the results of the classification report performed on the VGG-16 model. The model is performing very well on all classes except for classes 0 and O. This can be due to similar reasons

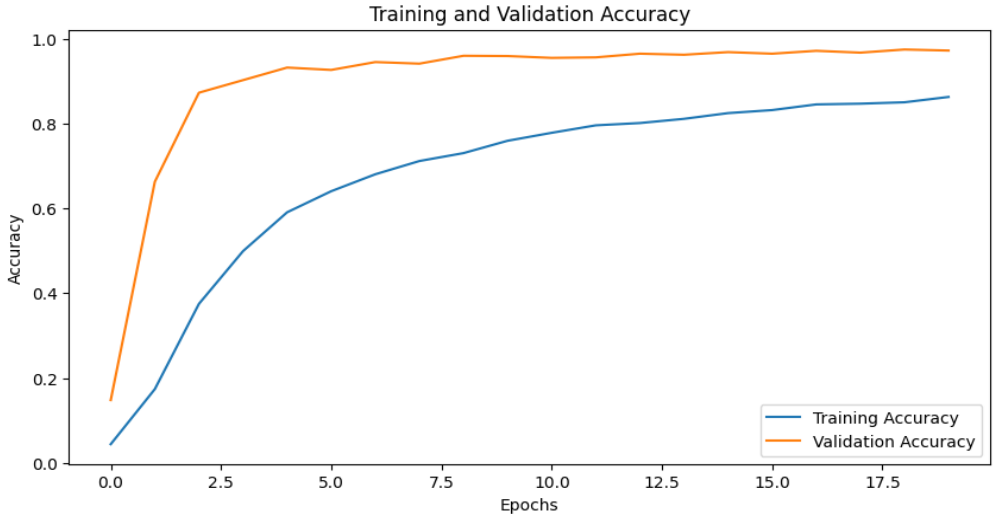

Figure 6: Training and Validation accuracy of my custom character recognition model plotted against the number of trained epochs.

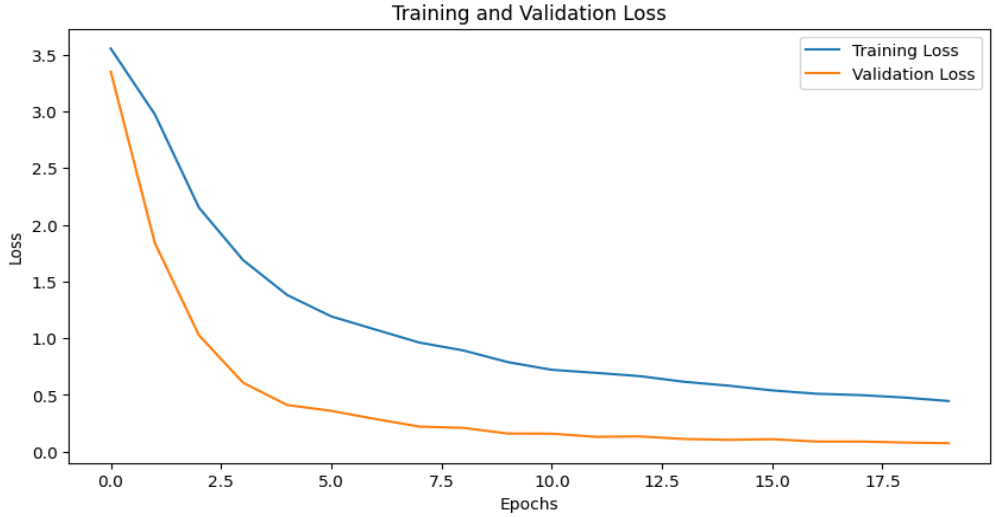

Figure 7: Training and Validation loss of my custom character recognition model plotted against the number of trained epochs.

that the custom character recognition model struggled with, such as difficulty discerning between 0 and O. Comparing the results side by side, it is evident that the custom model is performing better with class I, as class I's recall and F1-scores are 0.39 and 0.56 respectively, whereas the custom model's class I's recall and F1 scores are both 1.00. Class C is predicted better in the pre-trained model, with precision, recall and F1-scores of 0.92, 0.94 and 0.93 respectively. Comparing the overall metrics, the custom model performs better, with overall accuracy, macro averages and weighted averages for precision, recall and F1-score all at 0.97 while on the contrary, the overall accuracy, macro averages for precision, recall and F1-score and the weighted averages for recall and F1-score are at 0.94, with the weighted average for precision at 0.95.
Training accuracy and loss as well as validation accuracy and loss were plotted in the same manner as the custom model's results.

Figure 9 demonstrates the fluctuations in both training and validation accuracy across epochs throughout the model's training process. The training accuracy begins at 5%, and increases almost linearly to 55% by the fourth epoch. The model's accuracy increases at a slower rate, reaching 80% by the 10th epoch. The model then increases in accuracy at a slower rate, ending training with 90% accuracy by

Class	Metrics			Support
	Precision	Recall	F1-score	
0	0.68	0.41	0.51	32
1	0.76	1.00	0.86	34
2	1.00	1.00	1.00	32
3	1.00	0.94	0.97	35
4	1.00	0.97	0.99	35
5	1.00	0.94	0.97	35
6	1.00	0.97	0.98	33
7	1.00	1.00	1.00	34
8	0.95	1.00	0.97	36
9	1.00	0.95	0.97	39
A	1.00	1.00	1.00	37
B	0.97	0.97	0.97	36
C	0.92	0.94	0.93	35
D	1.00	0.97	0.98	33
E	0.97	0.91	0.94	32
F	0.91	0.97	0.94	31
G	0.88	0.88	0.88	33
H	1.00	1.00	1.00	35
I	1.00	0.39	0.56	28
J	0.82	0.96	0.89	28
K	1.00	0.87	0.93	31
L	0.97	1.00	0.99	33
M	1.00	0.97	0.99	34
N	0.94	1.00	0.97	34
O	0.55	0.82	0.66	28
P	1.00	1.00	1.00	35
Q	0.93	1.00	0.97	28
R	0.89	0.97	0.93	33
S	1.00	1.00	1.00	37
T	0.97	1.00	0.98	31
U	1.00	1.00	1.00	37
V	1.00	1.00	1.00	32
W	1.00	1.00	1.00	28
X	0.97	0.97	0.97	37
Y	0.94	1.00	0.97	33
Z	1.00	1.00	1.00	39
Overall Metrics				
Accuracy		0.94		1203
Macro Avg	0.94	0.94	0.94	1203
Weighted Avg	0.95	0.94	0.94	1203

Figure 8: Classification report of the VGG16 model with transfer learning.

the 28th epoch.
It is apparent that the validation accuracy begins at a higher percentage, with 26% accuracy, and quickly ascends to 76% accuracy by the first epoch. The model then reaches 85% by the second epoch, before reaching 90% accuracy by the fifth epoch. The model then slows down in its rate of increase in validation accuracy, levelling out and fluctuating around 94-97%, finishing training after 28 epochs at 95%.

The training and validation loss of the VGG-16 model is illustrated in figure 10. The training loss starts at 3.5, and quickly decreases to 2.0 by the third epoch. The rate of the training loss decreasing decreases after this, reaching 1.25 by the fifth epoch and 0.75 by the 10th epoch. The rate keeps decreasing and starts to plateau, reaching a loss of 0.45 at epoch 20 and finishing at a 0.45 loss at epoch 28.
The validation loss starts at 3.25, and swiftly decreases to 1.6 by the second epoch. By the fourth epoch, the validation loss is at 0.75. From the fourth epoch and beyond, the rate of validation loss decrease slows down significantly, almost flatlining at 0.25 by the 11th epoch and only fluctuating slightly until epoch 28, where the model completes training with a validation loss of 0.2.
The reason for the VGG-16 model ending training at 28 epochs is due to the EarlyStopping callback that was implemented during the model's implementation. The EarlyStopping callback halted training due to the validation loss not improving over three consecutive epochs.

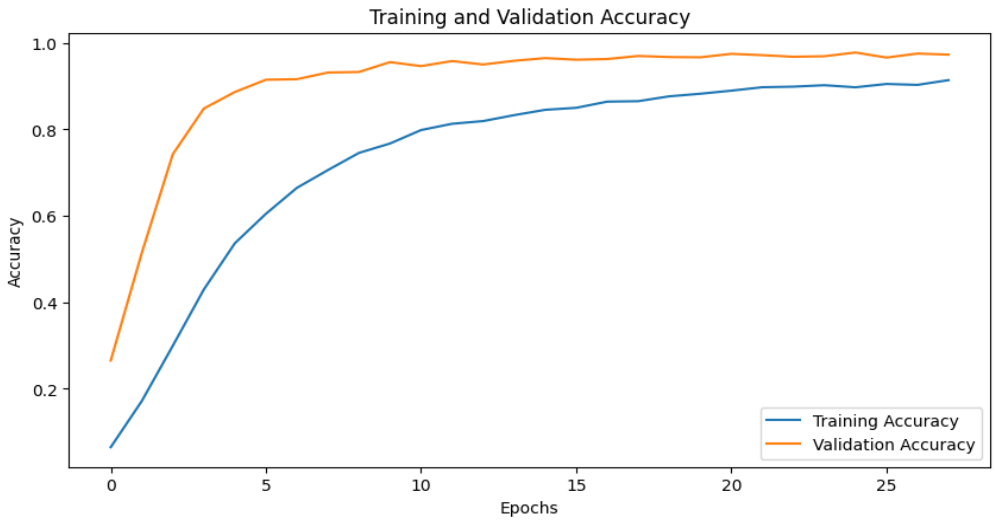

Figure 9: Training and Validation accuracy of the VGG-16 model trained with transfer learning plotted against the number of trained epochs.

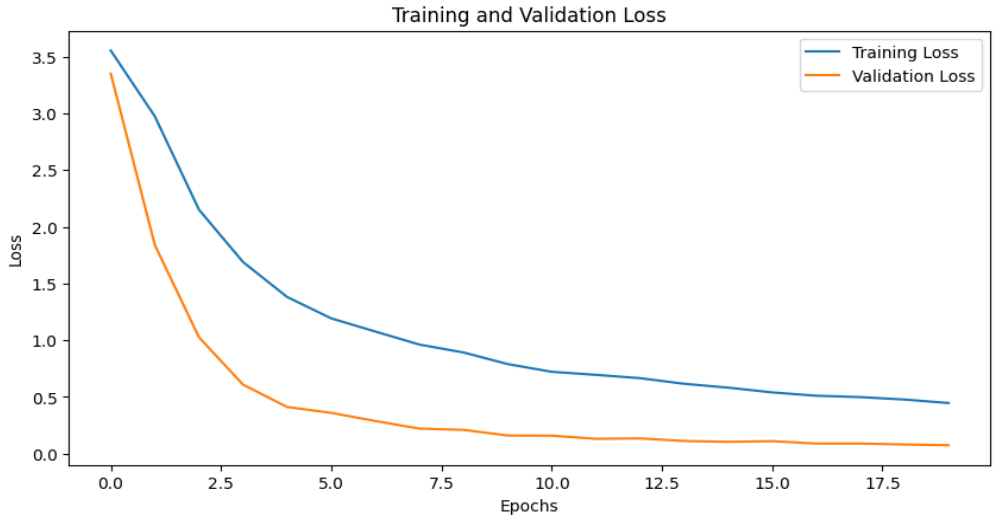

Figure 10: Training and Validation loss of the VGG-16 model trained with transfer learning plotted against the number of trained epochs.

5.3 Web Application

Prior to initiating testing for the web application, it was imperative to conduct a comprehensive analysis of the requirements. Since data was not being stored, database testing did not need to occur. Testing for the web application included functional testing and usability testing.

5.3.1 Functional Testing

Figure 11 displays the home/landing page of the ANPR app, and figure 12 displays the main page of the ANPR app. Functional testing was carried out by confirming the buttons worked as they should, ensuring navigation between pages occurred well, and ensuring user input validation when inputting images.

5.3.2 Usability Testing

Usability testing was performed by evaluating the user interface and ensuring it is user-friendly, clean, simple and intuitive. Additionally, I verified the placement of buttons and text elements within the user interface, ensuring optimal alignment, and confirmed that the font size was appropriate to maintain readability without being excessively large or small. .

Figure 11: Home page of the ANPR web app, located in the root path of the web app.

Figure 12: Main page of the ANPR web app, found in the /run-pipeline path.

5.4 Overview

The number plate detection and character segmentation algorithms perform best on lower quality images where the license plate is clearly visible with good lighting. The outcomes of the character recognition models indicate that employing a more complex architecture or leveraging transfer learning does not necessarily result in improved performance. It is a possibility that the complex architecture contributed to increased overfitting on the dataset used for training. Both models exhibit subpar performance when classifying instances belonging to classes '0' and 'O'. However, the custom model demonstrates superior performance in classifying instances of class 'I', whereas the pre-trained model outperforms the custom model in classifying instances of class 'C'. The web application underwent

both functional and usability testing procedures to validate its operational integrity and assess the user interface for continued user-friendliness.

5.5 Evaluation

After examining the results and what was developed, it is clear that the aims of the project have been met. One of the aims was to create a well-performing ANPR software with machine learning techniques. The character recognition model excels in all measured metrics, however the overall system demonstrates satisfactory performance. Its overall performance may fall short of initial expectations, mainly due to the license plate detection and character segmentation components. The second aim was developing a web application that provides the user with an interface to operate the ANPR software. This criteria has been met, by developing a Django web app allows for more scalability and simpler maintenance, though with a slightly harder setup than Flask.

6 Critical Assessment

All in all, the journey undertaken during this project had many challenges and lessons learned, but created an outcome that has multiple strengths and some weaknesses. Some strengths observed during this project include the ease of accessing data for the character recognition model and the abundance of online resources available to provide support when encountering challenges. However, the lack of computational resources at times as well as subpar judgment of the time that would have been necessary to produce a better project are a few of the challenges and weaknesses faced. Additional strengths of the project include its user-friendly interface and the efficacy of the character recognition model. Contrarily, weaknesses are evident in the number plate detection and character segmentation components. Potential areas for improvement within the project include assigning more time and creating models for the number plate detection & character segmentation components as well as adding extra features to the app. Some lessons learned comprise better time management and further background research into different, more complex architectures.

7 Conclusion

In conclusion, the main objectives of this project was to create an ANPR system utilising machine learning and incorporate that into a web application to allow users to interact with the ANPR system via a user interface. The literature review gave insight into the foundations of ANPR technology. Additionally, the project specification gave information on the approach that was to be taken towards the project and a basic outline on what was to be expected. The design section established the three key components to the ANPR system and the web application, The number plate detection and character segmentation components were both developed using OpenCV and NumPy, and the character recognition model was developed using TensorFlow. In addition to this, the development section delved into the points discussed in the design section in more detail, examining how each component works in detail and analysing the decisions made during the process, such as opting to not use Google Cloud's App Engine and thus not hosting the web application, due to the steep financial demands to start and maintain a web application publicly. The results section displayed the data obtained from tests and evaluated upon the data, explaining the results and what caused these results. Gathering from the results, it is clear that the custom model performs better than the pre-trained model due to scoring better overall accuracy, higher macro averages for precision, recall and F1-score as well as higher weighted averages for precision, recall and F1-score over the pre-trained model.
Overall, this project has been successful, despite the challenges. The project has potential to be expanded upon and improved on, and the development process has built upon my coding skills, especially with machine learning.

References

[1] Feb 2023. URL https://www.softwebsolutions.com/resources/number-plate-recognition-using-computer-vision.html.

[2] Sing T Bow. *Pattern recognition and image preprocessing*. CRC press, 2002.

[3] Arun Chand, S. Jayesh, and A.B. Bhasi. Road traffic accidents: An overview of data sources, analysis techniques and contributing factors. *Materials Today: Proceedings*, 47:5135–5141, 2021. ISSN 2214-7853. doi: https://doi.org/10.1016/j.matpr.2021.05.415. URL https://www.sciencedirect.com/science/article/pii/S2214785321040153. International Conference on Sustainable materials, Manufacturing and Renewable Technologies 2021.

[4] Rahul Chauhan, Kamal Kumar Ghanshala, and RC Joshi. Convolutional neural network (cnn) for image detection and recognition. In *2018 first international conference on secure cyber computing and communication (ICSCCC)*, pages 278–282. IEEE, 2018.

[5] Issam El Naqa and Martin J Murphy. *What is machine learning?* Springer, 2015.

[6] Jeff Forcier, Paul Bissex, and Wesley J Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.

[7] GeeksforGeeks. Vgg-16: Cnn model, Mar 2024. URL https://www.geeksforgeeks.org/vgg-16-cnn-model/.

[8] Devndra Ghimire. Comparative study on python web frameworks: Flask and django. 2020.

[9] Hayder Hasan, Helmi ZM Shafri, and Mohammed Habshi. A comparison between support vector machine (svm) and convolutional neural network (cnn) models for hyperspectral image classification. In *IOP Conference Series: Earth and Environmental Science*, volume 357, page 012035. IOP Publishing, 2019.

[10] Thomas Heseltine, Nick Pears, and Jim Austin. Evaluation of image preprocessing techniques for eigenface-based face recognition. In *Second International Conference on Image and Graphics*, volume 4875, pages 677–685. Spie, 2002.

[11] Alok Nikhil Jha, Niladri Chatterjee, and Geetam Tiwari. A performance analysis of prediction techniques for impacting vehicles in hit-and-run road accidents. *Accident Analysis Prevention*, 157:106164, 2021. ISSN 0001-4575. doi: https://doi.org/10.1016/j.aap.2021.106164. URL https://www.sciencedirect.com/science/article/pii/S0001457521001950.

[12] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[13] Irfan Khan and Nida Aslam. A deep-learning-based framework for automated diagnosis of covid-19 using x-ray images. *Information*, 11:419, 08 2020. doi: 10.3390/info11090419.

[14] Barath Kumar. Image preprocessing - why is it necessary?, Feb 2021. URL https://medium.com/spidernitt/image-preprocessing-why-is-it-necessary-8895b8b08c1d.

[15] R.A. Lotufo, A.D. Morgan, and A.S. Johnson. Automatic number-plate recognition. In *IEE Colloquium on Image Analysis for Transport Applications*, pages 6/1–6/6, 1990.

[16] Ravina Mithe, Supriya Indalkar, and Nilam Divekar. Optical character recognition. *International journal of recent technology and engineering (IJRTE)*, 2(1):72–75, 2013.

[17] Kuntal Kumar Pal and KS Sudeep. Preprocessing for image classification by convolutional neural networks. In *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 1778–1781. IEEE, 2016.

[18] Chirag Patel, Dipti Shah, and Atul Patel. Automatic number plate recognition system (anpr): A survey. *International Journal of Computer Applications*, 69(9), 2013.

[19] Jianxin Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 5(23):495, 2017.