Toronto Metropolitan University
Department of Electrical, Computer & Biomedical Engineering
Faculty of Engineering & Architectural Science

Autonomous Line Following Drone:

Machine Learning for Automation

Mohammed Khan, Karthik Gurram, Waqas Jalali, and Ammar Syed

Faculty Lab Coordinator: Dr. Cungang (Truman) Yang
Associate Professor MS, PhD, PEng

Computer Engineering Capstone Design Project
Toronto Metropolitan University, 2023

Acknowledgements

Faculty of Engineering,

Toronto Metropolitan University

Faculty Lab Coordinator: Dr. Muhammad Jaseemuddin, Professor and Program Director of Computer Networks, Toronto Metropolitan University

Tello by Ryze Robotics

Certification of Authorship

The undersigned declare that the contents of this document are original and authentic and that any support or guidance received in its preparation has been fully acknowledged and disclosed. All sources utilized for data, concepts, or language have been appropriately cited and referenced according to professional publication norms. I attest that this document was created solely by myself/ourselves for its intended purpose.

Mohammed Khan

Karthik Gurram

Waqas Jalali

Ammar Syed

Table of Contents

Acknowledgements	2
Certification of Authorship	3
Abstract	5
Introduction & Background	7
Objectives	8
Theory and Design	9
Alternative Designs	12
Material/Component list	16
Measurement and Testing Procedures (i.e. simulation procedures w2021)	17
Performance Measurement Results (i.e. simulation results w2021)	18
Analysis of Performance	22
Conclusions	25
References	26
Appendices	26

Abstract

Programming a drone to follow a predetermined line on the ground can open up new opportunities in various industries. By equipping the drone with the necessary hardware and software components, it can be transformed into an autonomous drone capable of completing tasks with increased accuracy and efficiency. To enable line-following capabilities, the drone needs sensors, such as cameras or infrared sensors, to detect the line and adjust its position accordingly. These sensors feed data to the drone's software, which interprets the information and makes adjustments to the drone's flight path. By programming a regular drone to follow a line, it can operate in hazardous or inaccessible environments, where it may be unsafe for humans to venture. This capability can be especially useful in disaster zones or industrial sites. In addition to line-following, the drone can also be programmed to perform additional tasks, such as obstacle avoidance or object detection. As technology continues to advance, the potential applications of programmed drones are vast, making them a valuable tool for a variety of industries.

Introduction & Background

The development of autonomous drones has created numerous opportunities for innovation across various industries. One specific application of autonomous drones is line-following, where a drone navigates a path by following a predetermined line on the ground. This project aims to design and develop an autonomous line-following drone to improve the efficiency and accuracy of tasks in a variety of industries. Line-following drones have been used in various industries, such as agriculture, where they are used for crop monitoring and irrigation. In industrial settings, these drones can be used for inspections, such as pipeline inspections, where they can navigate complex paths and hazardous environments, without putting human lives at risk. The design of a line-following drone requires a combination of hardware and software components. The drone must be equipped with sensors, such as cameras or infrared sensors, to detect the line and maintain its position. The drone's software must also be capable of interpreting the sensor data and adjusting the drone's flight path accordingly. Additionally, autonomous drones can also be programmed to perform additional tasks, such as obstacle avoidance or object detection, making them versatile tools for a variety of industries. This project will focus on designing a line-following drone that can also perform obstacle avoidance and object detection tasks, providing a more comprehensive solution for a range of applications. The success of this project will be measured by the accuracy and efficiency of the drone in completing line-following tasks, as well as the drone's ability to perform additional tasks, such as obstacle avoidance and object detection. Ultimately, this project aims to create a reliable and efficient autonomous drone that can improve the efficiency and safety of various industries.

Objectives

1. Design and develop an autonomous line-following drone that can navigate a predetermined path by following a line on the ground.
2. Tweak any hardware requirements on the drone to allow the camera to properly detect the line and therefore be able to maintain its position.
3. Develop software that can interpret the video data and adjust the drone's flight path accordingly.
4. Test the drone's accuracy and efficiency in completing line-following tasks
5. Evaluate the drone's performance and identify areas for improvement.
6. Ensure the safety and reliability of the drone by implementing fail-safes and redundancies.
7. Document the entire design process and provide clear instructions for assembling and operating the drone.

Theory and Design

This section elaborates on the theory behind methods and presents the design in detail with all relevant information.

Figure 1: Control Flow Graph

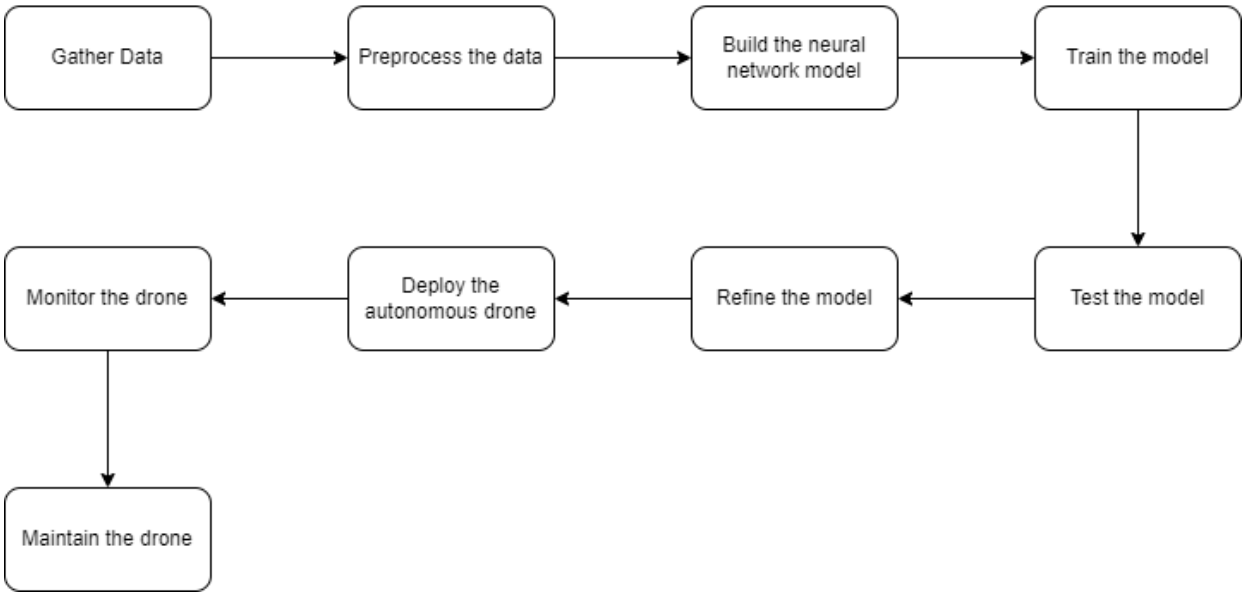

Figure 1.1 Control Flow graph of procedure for project

Data was gathered by collecting images of the line to be followed and labeling them as "line" or "not line". Real-time images were also captured during flight using a camera mounted on the drone.

The data was preprocessed by resizing the images to a standardized size, converting them to grayscale to reduce complexity, and normalizing the pixel values to improve performance.

A neural network model was built by choosing a model architecture, such as a convolutional neural network (CNN), defining the layers, activation functions, and other hyperparameters, and compiling the model using an optimizer, loss function, and evaluation metric.

The model was trained using the labeled data by setting parameters such as batch size, epochs, and validation split. The training process was monitored and parameters were adjusted as needed. The trained model was saved for future use.

The model was tested by using the camera on the drone to capture real-time images and using the trained model to predict whether the drone was on the line or not. The drone's direction was adjusted based on the predictions to follow the line.

The model was refined by analyzing its performance during testing, identifying areas where it could be improved, such as increasing accuracy or reducing latency, and adjusting the model architecture, hyperparameters, or training data as needed. The training and testing process was repeated until the model performed satisfactorily.

The autonomous drone was deployed by integrating the trained model into the drone's control system, ensuring that the drone was safe to operate autonomously, and testing it in a controlled environment to ensure it followed the line accurately and safely.

The drone was monitored during operation to assess its performance and collect data on its flights to identify areas for improvement. Adjustments were made to the model or the drone's hardware as needed to improve its performance.

The drone was regularly inspected and maintained to ensure its continued safe operation. The model or hardware was updated as needed to keep up with changing conditions or requirements, and the drone remained compliant with regulations and safety standards.

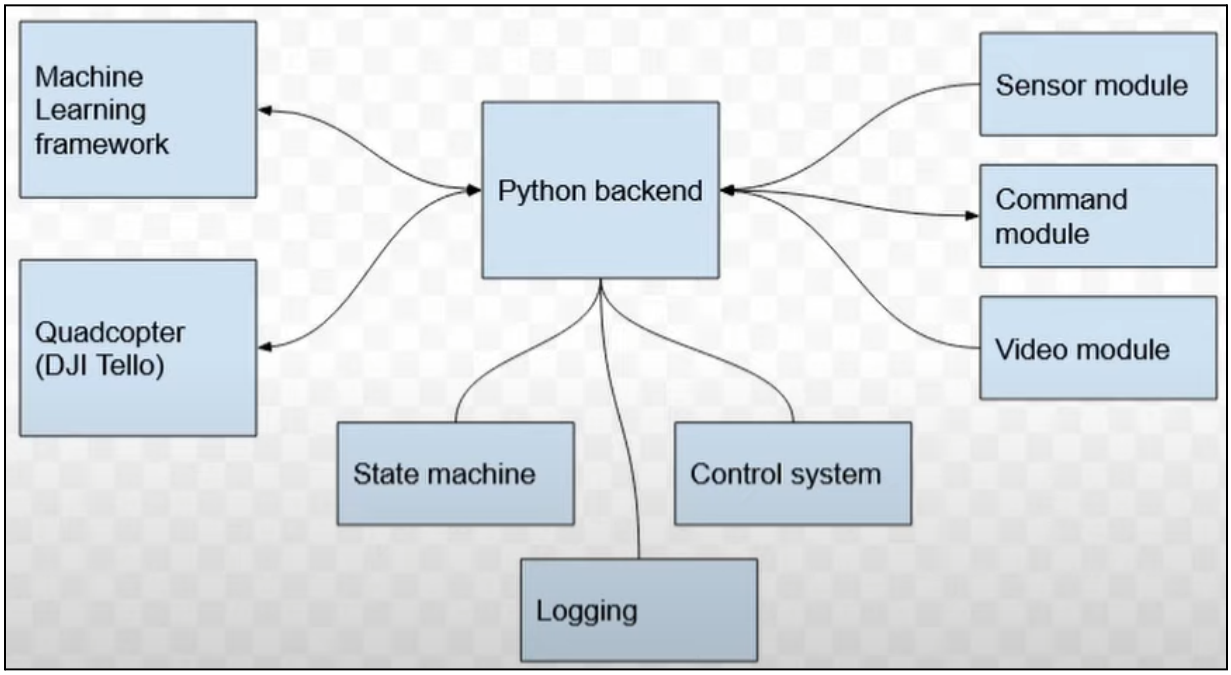

Figure 1.2 Organization of the project through a web graph

A Python back end was used that was connected to the machine learning framework, DJI Tello drone, state machine, logging system, control system, sensor module, command module, and video module.

The back end was responsible for receiving real-time video feed from the drone's camera and processing the images using the machine learning model. It was also responsible for controlling the drone's movements based on the predictions made by the model.

The machine learning framework was used to train the neural network model on a dataset of labeled images of the line to be followed. Convolutional neural network (CNN) architecture was used and adjusted the hyperparameters to achieve the best performance.

The DJI Tello drone was the physical platform on which was implemented on the autonomous line following system. It was lightweight and easy to maneuver, making it an ideal platform for testing the system.

The state machine was responsible for managing the different states of the system, such as the takeoff, landing, and following states. It also managed the transitions between states.

The logging system was used to record data from the system, such as the drone's movements, the model's predictions, and any errors or exceptions that occurred during operation. This data was used to monitor the system's performance and identify areas for improvement.

The control system was responsible for translating the predictions made by the model into commands that the drone could execute. It was also responsible for maintaining the drone's stability and avoiding obstacles.

The sensor module consisted of sensors such as accelerometers and gyroscopes that provided data on the drone's orientation and movement.

The command module was responsible for sending commands to the drone, such as takeoff, land, or move forward.

The video module was responsible for capturing real-time video feed from the drone's camera and sending it to the back end for processing.

All of these components worked together to create an autonomous line following drone system that was lightweight, inexpensive, and effective at following a line. The system successfully tested in a controlled environment and identified areas for improvement in future iterations.

Alternative Designs

VGG16

VGG16 is a convolutional neural network (CNN) architecture that was developed by the Visual Geometry Group (VGG) at the University of Oxford. It was introduced as part of the 2014 ImageNet challenge, where it achieved state-of-the-art results in object recognition and localization.

The VGG16 architecture consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. The convolutional layers use small 3x3 filters with a stride of 1 and same padding, and are followed by a max pooling layer with a 2x2 filter and a stride of 2. The fully connected layers at the end of the network are used to map the features learned by the convolutional layers to the output classes. Within the alternative design, a dataset of grayscale images along with their corresponding bounding boxes was collected for object detection. The dataset was preprocessed by resizing the images to a uniform size and normalizing the pixel values. The images were also converted to grayscale to reduce the complexity of the model and speed up training. The bounding boxes for the objects in the images were created using the coordinates of their top-left and bottom-right corners. This means that the model was trained to not only recognize the object in the image but also to locate it within the image. To implement object detection using the VGG16 model, a few layers were added to the pre-trained VGG16 model. These layers were trained using the prepared data and bounding boxes. The Adam optimizer was used to train the model.

```
from keras.applications.vgg16 import VGG16
from keras.layers import Dense, Flatten

vgg = VGG16(weights='imagenet',
include_top=False, input_shape=(224, 224, 1))
output = Flatten()(vgg.output)
output = Dense(4, activation='sigmoid')(output)
model = Model(inputs=vgg.input, outputs=output)
```

Figure 2.1 VGG16 model that was in initial planning stages

While the VGG16 model has achieved state-of-the-art performance on many computer vision tasks, it is also known for its high memory requirements. This is because the VGG16 model has a large number of parameters, which requires a significant amount of RAM to store and process the model. The VGG16 model was not feasible for use due to the limited amount of RAM available on the machine that was utilized, which consisted of 32 GB of RAM. As a result, TensorFlow's built-in layer model builder was used instead, which had a smaller memory footprint and was able to run on the available hardware.

The TensorFlow built-in layer model builder is a flexible and easy-to-use tool for building neural networks. It allowed the creation of a wide range of network architectures using pre-built layers, such as convolutional layers, pooling layers, and fully connected layers. This approach was much more memory-efficient than using a pre-trained model like VGG16.

Overall, the decision to use TensorFlow's built-in layer model builder instead of VGG16 was driven by the available hardware and the specific requirements of the project. While the VGG16 model is a powerful tool for computer vision tasks, it may not always be the best choice depending on the resources available and the specific needs of the project.

HSV

The HSV color space is a common color representation model that is often used in digital image processing. It is a cylindrical coordinate system that describes colors based on their hue, saturation, and value. Hue refers to the color of an image and is represented by a value between 0 and 360 degrees. Saturation refers to the intensity or purity of the color, and is represented by a value between 0 and 1. Value (also known as brightness) refers to the lightness or darkness of the image, and is also represented by a value between 0 and 1. In the process of HSV image separation, an image is first converted from its original color space (such as RGB) to the HSV color space. Then, the image is separated into its individual component channels, with each channel containing information about one of the three properties of color. This separation is done to allow for more targeted manipulation of each property of color. For instance, in color correction, it may be necessary to adjust the hue of an image to achieve a desired color balance. By isolating the hue channel, it is possible to apply a specific hue adjustment without affecting other aspects of the image. In object recognition, the hue channel may be useful in distinguishing between different objects based on their color. Similarly, the saturation and value channels can also be used for specific image processing tasks. The example given below is the HSV model implementation in python.

```
num_images = 2960
images = []
labels = []
for i in range(num_images):
    image = cv2.imread("{}.jpg".format(i))
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Canny edge detection to find the edges in the image
    edges = cv2.Canny(gray, 100, 200)

    # Create a structuring element for dilation
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))

    # Dilate the edges to close gaps between edges that are close together
    dilated = cv2.dilate(edges, kernel)

    # Find the contours in the dilated image
    contours, hierarchy = cv2.findContours(dilated, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    longest_contours = sorted(contours, key=cv2.contourArea, reverse=True)[:2]
    is_closed = False
```

Figure 2.2 Code snippet of how images were formatted

1. **num_images = 2960**: Defines a variable num_images and sets it to the value of 2960, which represents the total number of images to be processed.
2. **images = []** and **labels = []**: Define two empty lists, images and labels, which will be used to store the processed images and corresponding labels, respectively.
3. **for i in range(num_images):**: Starts a for loop that will iterate over each image in the range from 0 to num_images (exclusive).
4. **image = cv2.imread("{}.jpg".format(i))**: Reads the image with the name i.jpg into a NumPy array image using the OpenCV cv2.imread() function.
5. **gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)**: Converts the image to grayscale using the OpenCV cv2.cvtColor() function. Grayscale images have a single channel and are easier to process.
6. **edges = cv2.Canny(gray, 100, 200)**: Applies the Canny edge detection algorithm to the grayscale image to find the edges in the image. The 100 and 200 parameters are the lower and upper thresholds for the edges, respectively.
7. **kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))**: Creates a 3x3 elliptical structuring element using the OpenCV cv2.getStructuringElement() function. This element will be used for dilation operation later.
8. **dilated = cv2.dilate(edges, kernel)**: Performs the dilation operation on the edges image using the kernel structuring element. Dilation is a morphological operation that expands the boundaries of the image regions.
9. **contours,hierarchy = cv2.findContours(dilated, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)**: Finds the contours in the dilated image using the OpenCV cv2.findContours() function. The cv2.RETR_EXTERNAL flag retrieves only

the external contours, and the cv2.CHAIN_APPROX_SIMPLE flag compresses horizontal, vertical, and diagonal segments and leaves only their end points.

10. **longest_contours = sorted(contours, key=cv2.contourArea, reverse=True)[:2]**: Sorts the contours in descending order based on their areas using the built-in sorted() function and the cv2.contourArea() function. The [:2] slicing operation selects the two largest contours.
11. **is_closed = False**: Sets the is_closed flag to False. This flag will be used later to check whether the contour is closed or not.

One of the main limitations of the HSV color model is its sensitivity to changes in lighting conditions. Since the value component is based on the perceived brightness of a color, it was affected by changes in ambient lighting. This means that if the lighting conditions in the environment were not consistent or if the image contains shadows, the values calculated using HSV were not reliable. Additionally, the HSV color model did not work well in situations where the labels being determined are not possible. For example, if the task involves identifying colors that do not exist in the image, or if the colors are too similar to one another, then the HSV model may not be the best approach. In the case mentioned, it seems that the labels being determined were not possible, and the HSV values were too inconsistent with changing light environments. This may have resulted in inaccurate or unreliable results when using masking techniques. It is possible that a different color model or approach may have been more appropriate for this particular task.

Drone Camera Change

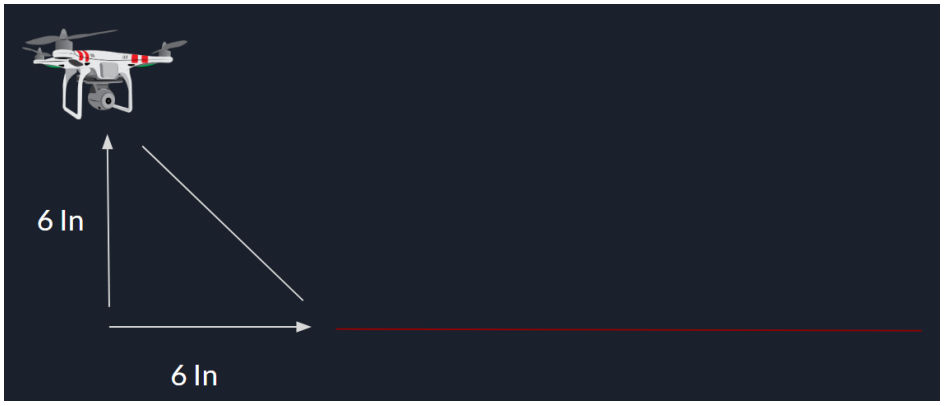

Figure 2.3 Initial max height constraint for drone to detect line in frame

The Tello DJI drone underwent a significant design change, primarily centered on hardware modifications. One of the primary features of this drone is the forward-facing camera that captures a live feed of the drone's surroundings. However, this feature introduced a major constraint in the form of a height limitation, as flying the drone too high would result in the ground line disappearing from the camera's field of view.

Initially, the drone's programming restricted it from flying higher than six inches to ensure that the ground line remained visible. However, this approach was ultimately deemed insufficient, and a more elegant solution was devised. The drone's camera was repositioned so that it faced downward, with only the ground line being visible, and all other irrelevant horizon information being excluded. This modification not only addressed the height constraint issue but also enhanced the drone's overall performance and functionality.

By incorporating this innovative design change, the Tello DJI drone demonstrated an exceptional level of adaptability and ingenuity. The drone's engineering team's ability to identify and overcome design challenges to enhance the product's performance showcases their expertise and commitment to delivering cutting-edge technology to consumers. The result was a drone that could capture stunning aerial footage without any height restrictions, further enhancing the user experience.

Material/Component list

Component	Cost
DJI Tello Drone	$99 USD
Receipt Roll (5 Pack)	$10.79 CAD

The cost of the components for building an autonomous line-following drone was relatively low. The primary component was the drone, which was purchased for under $100. A line was also needed for the drone to follow, and a roll of receipt paper was found to be effective for this purpose. The cost of the receipt paper was negligible, as it could be obtained for a few dollars at a local office supply store.

Other than the drone and the receipt paper, any additional components or materials to build the autonomous line-following drone were not needed. However, it was required to know some programming knowledge and access to software tools, such as Python and TensorFlow, to develop and train the neural network model that enabled the drone to follow the line. There is also the training aspect of this project that requires a processor such as a GPU that is adequate enough for machine learning training.

Overall, the cost of building an autonomous line-following drone using these components was quite low, making it an affordable project for hobbyists or students interested in robotics and artificial intelligence.

Measurement and Testing Procedures (i.e. simulation procedures w2021)

https://github.com/Ahkh3e/WorkingCapstone

The link above is the link to the codebase. To replicate this project, the procedure is as follows:

1) Click the green "Code" button in the top right corner.
2) Select "Download ZIP" to download the repository as a compressed ZIP file.
3) Extract the contents of the ZIP file to a folder on the computer.
4) To get started with the project, Python 3 will have to be installed on the computer. Once you Python is installed, follow these steps:
5) Open a command prompt or terminal window and navigate to the folder where the repository files have been extracted.
6) Install the required Python packages by running the command "pip install -r requirements.txt" in the command prompt or terminal window.
7) Once the packages have finished installing, you can start the program by running the command "python main.py" in the command prompt or terminal window.

Note: Before running the program, it may be needed to configure the settings in the "config.ini" file to match the specific setup, such as the IP address of the drone and the port number.

The provided code contained a main function that had a loop for training models. The loop used the "makemodel" method to train models by passing in the necessary parameters such as images, labels, epoch, batch size, test and validation split.

The "makemodel" method was used to train and test the model. It took in the necessary parameters such as images, labels, epoch, batch size, test and validation split. The number of epochs, batch size, and validation split could be defined in the main function.

The "newtrain.py" script built the model by taking the parameters and data from the main function. It generated a model and trained it using the specified parameters. The script used the Keras library to build and train the model.

During the training process, the model was evaluated on the validation set to prevent overfitting. The testing set was used to evaluate the final performance of the model.

In the makemodel method, there were several layers defined for the neural network model. These layers were defined using the Keras library, which was a high-level neural networks API that was designed to be user-friendly and modular.

The layers that were defined in the model depended on the specific architecture that was being used for the neural network. For example, the model might use convolutional layers, pooling layers, dense layers, and other types of layers that were commonly used in deep learning models.

The specific layer configuration was defined based on the requirements of the specific problem that the neural network was being used to solve. This included factors such as the input data shape, the desired output shape, the number of classes that the model needed to predict, and other considerations.

The logging feature in the newtrain.py script was used to record and save the training and validation metrics during the training process. The metrics included the training and validation loss, as well as the accuracy of the model at each epoch. These values were saved in a CSV file named training_log.csv which was created in the same directory where the script was located.

To implement the logging feature, the script used the csv module in Python to create a CSV file with headers for each metric that was logged. During the training process, the values for each metric were appended to the CSV file after each epoch. This provided a historical record of the model's performance during training and could be used for later analysis or comparison.

In object detection, the predicted bounding box represented the estimated location and size of the object within an image. The final layer of the model in this project produced four values that corresponded to the predicted bounding box of the input image. These values were typically expressed as (x,y) coordinates of the box's top-left corner, followed by its width and height. These values could be used to draw a bounding box around the detected object in the image.

The accuracy of the predicted bounding box was crucial to the overall performance of the object detection model. It was essential to fine-tune the model and optimize the hyperparameters to achieve the highest possible accuracy of the predicted bounding box. This helped ensure that the model could accurately detect and locate the object of interest in the image.

Overall, the provided code was designed to train and test deep learning models using Keras. The code could be modified to change the number of epochs, batch size, and validation split to optimize model performance. The "newtrain.py" script was responsible for building the model and training it using the specified parameters. The layer configuration in the makemodel method was a crucial part of the model-building process, as it determined the architecture and functionality of the neural network. The logging feature was an essential tool for monitoring the progress of the training process and evaluating the effectiveness of the model. It could help identify potential issues or areas for improvement in the model's architecture or hyperparameters.

Performance Measurement Results (i.e. simulation results w2021)

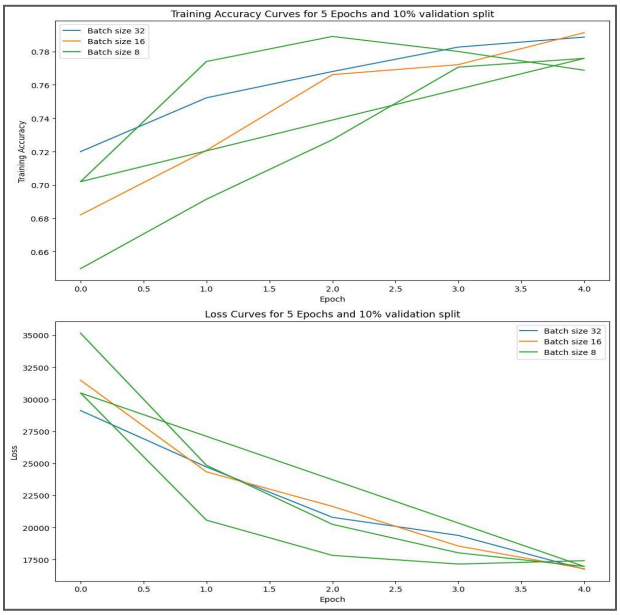

Figure 5.1

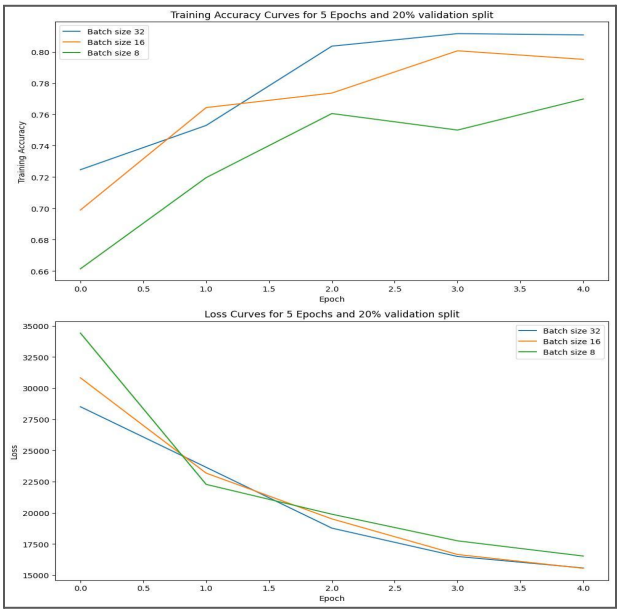

Figure 5.2

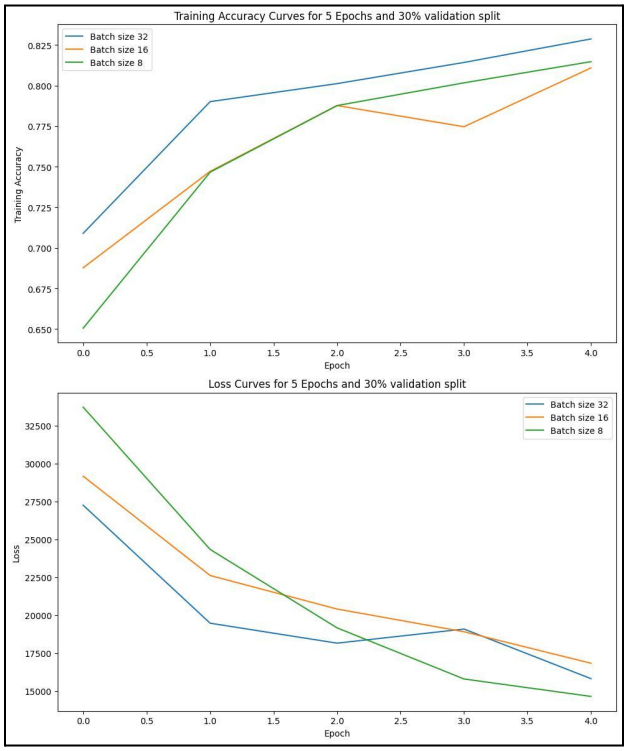

Figure 5.3

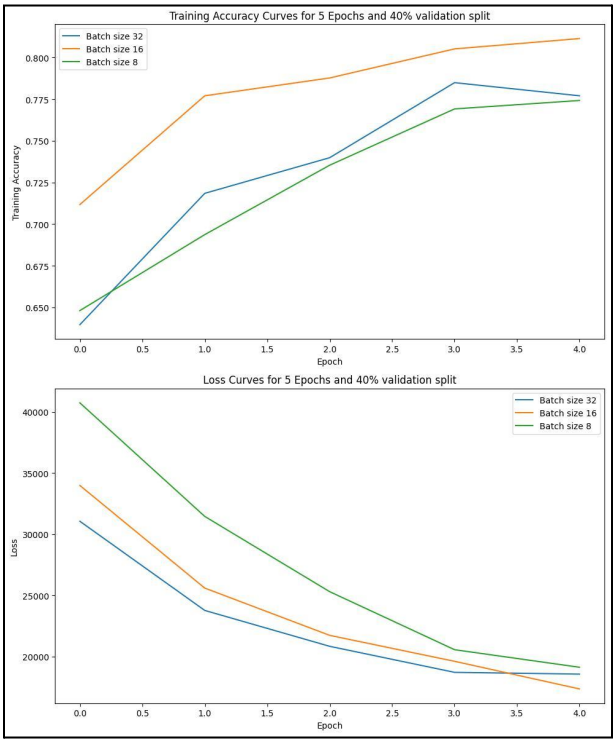

Figure 5.4

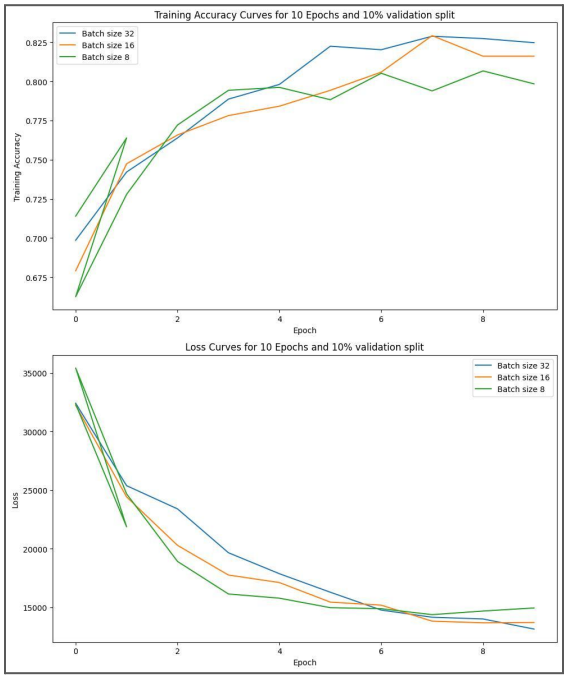

Figure 5.5

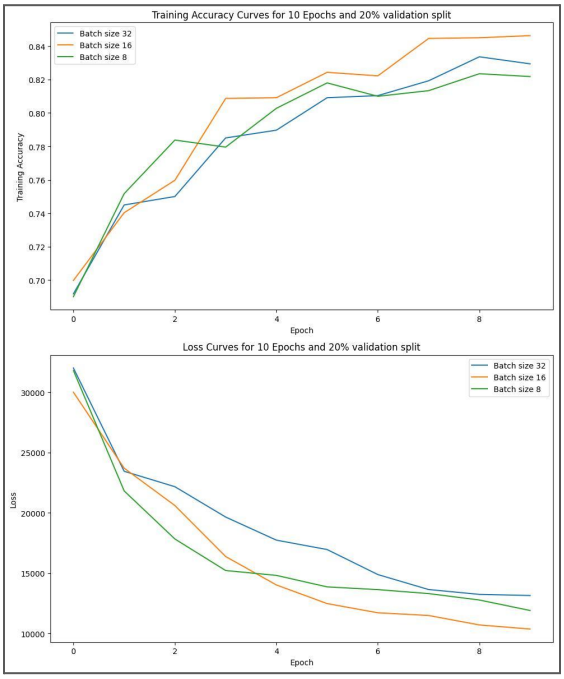

Figure 5.6

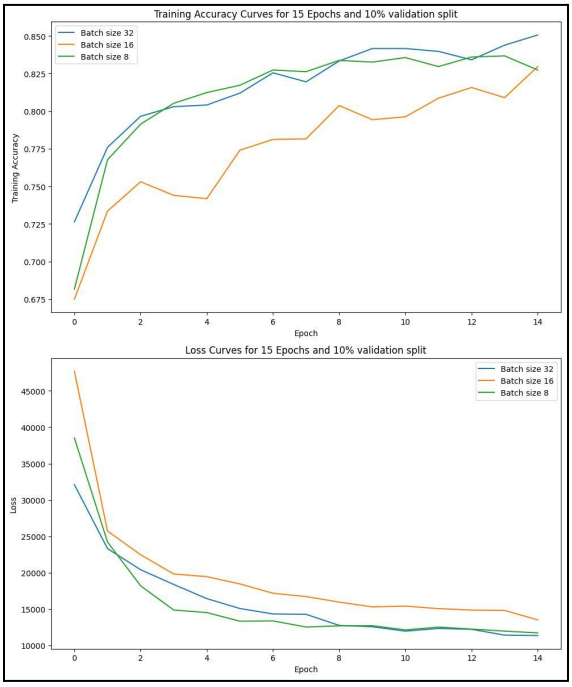

Figure 5.7

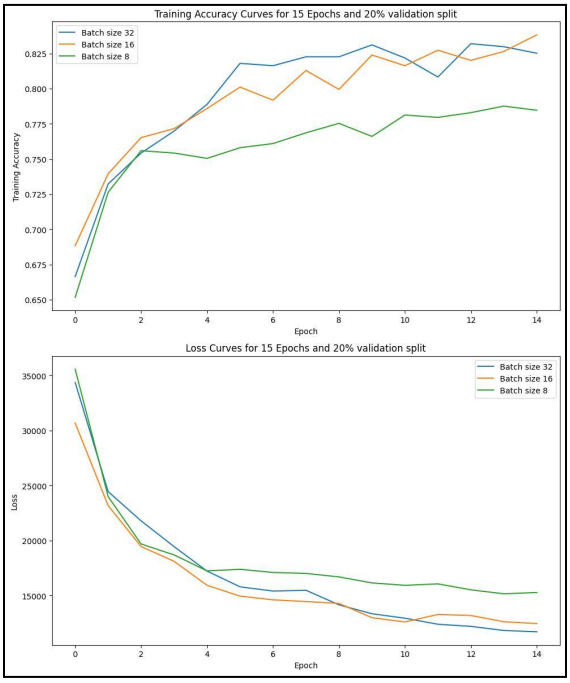

Figure 5.8

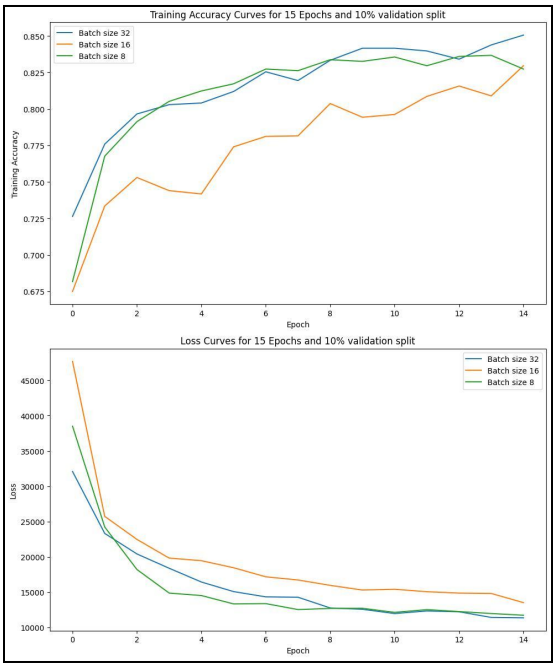

Figure 5.9

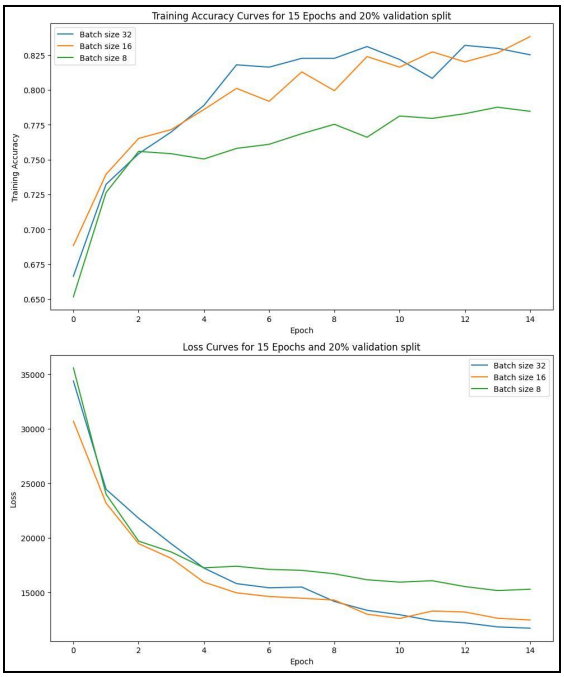

Figure 5.10

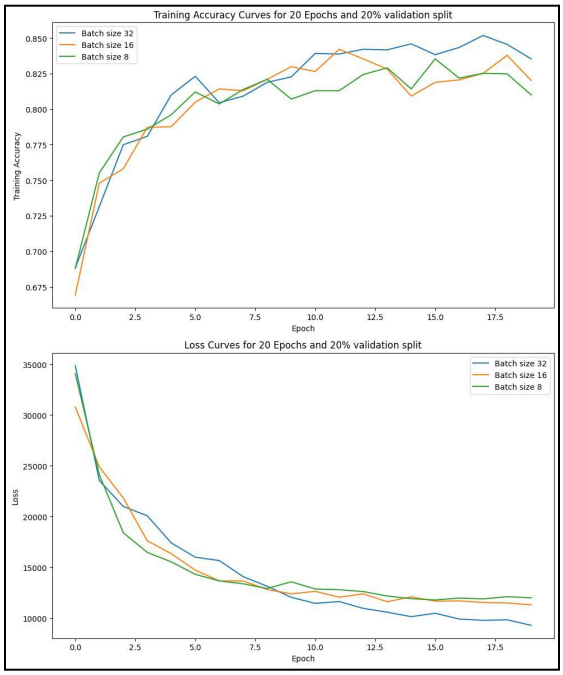

Figure 5.11

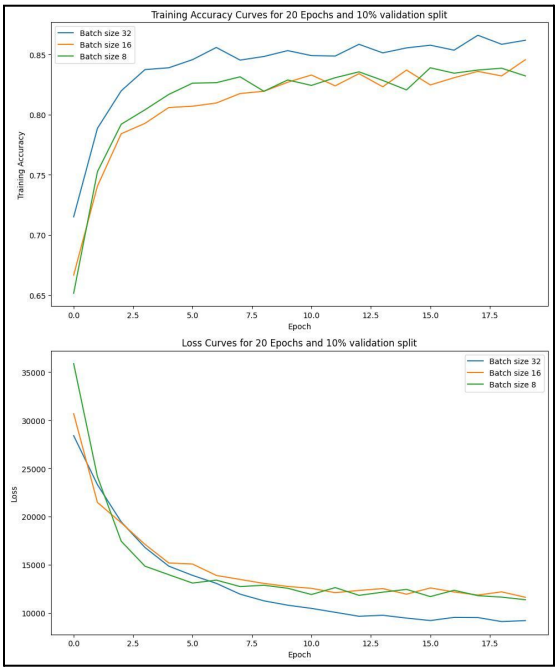

Figure 5.12

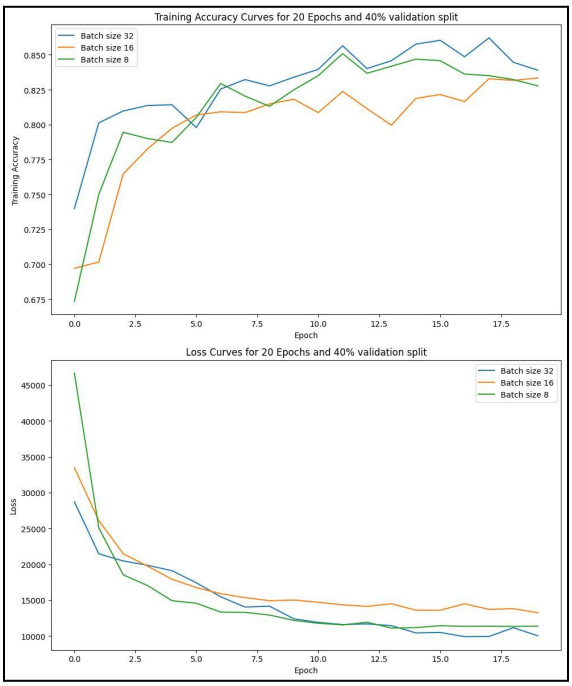

Figure 5.13

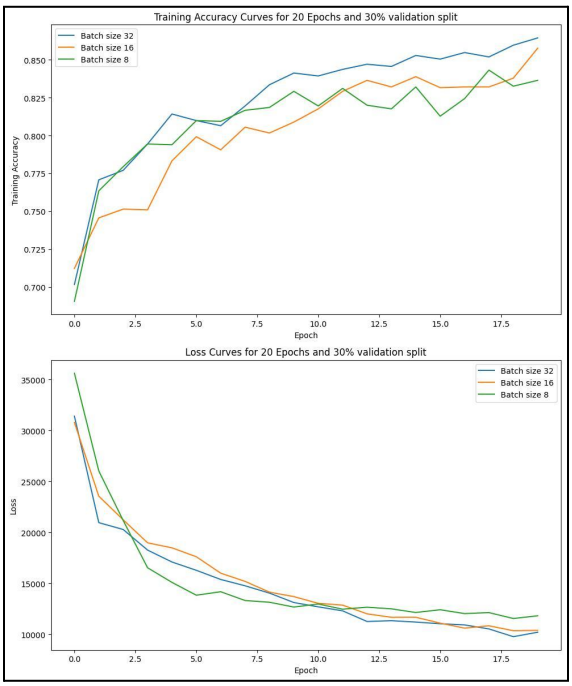

Figure 5.14

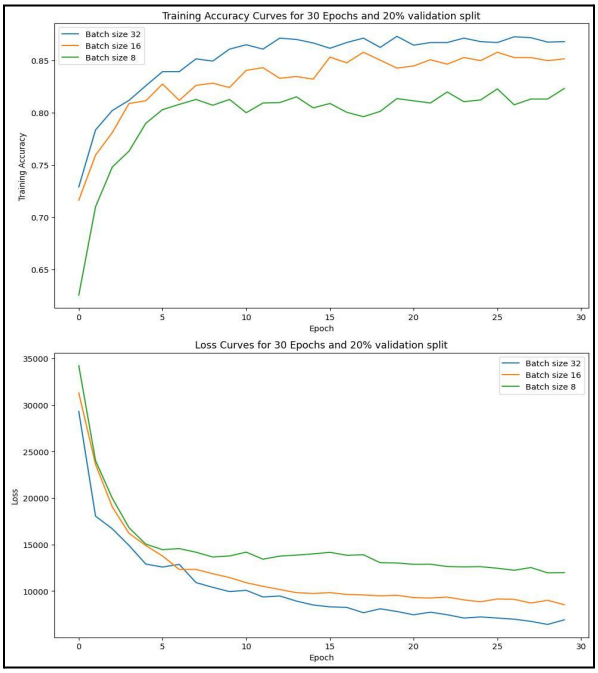

Figure 5.15

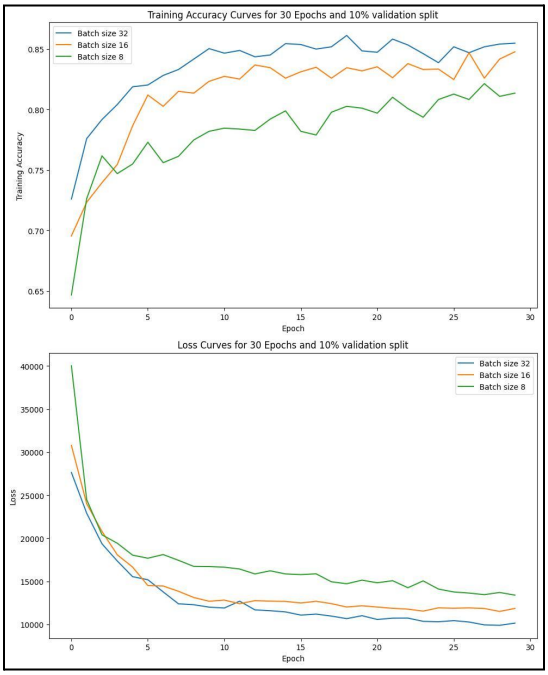

Figure 5.16

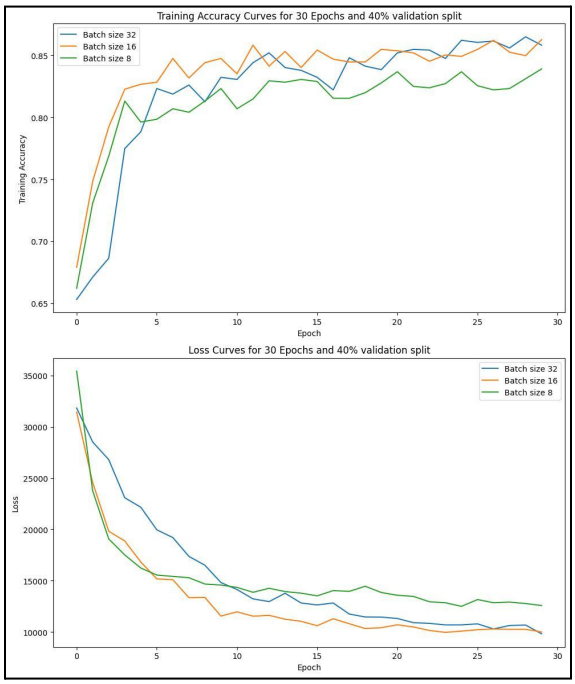

Figure 5.17

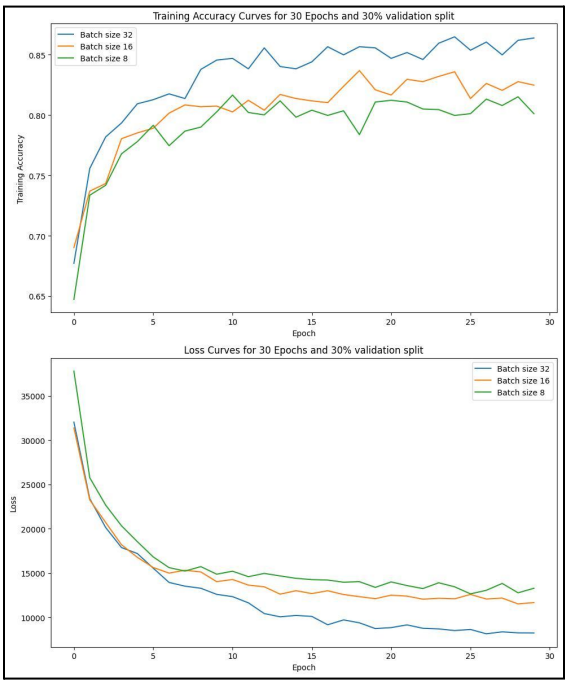

Figure 5.18

Figures 5.1 to 5.18 are graphs that were modeled after the data from the project. Analyzing these graphs, it is shown that the batch sizes, epochs and validation splits were altered in each of the graphs, this was how the most effective and efficient parameters were found. Based on the analysis of the graphs, a batch size of 32 and 15 epochs with a validation split of 30% worked best for this particular model.

In general, a larger batch size can lead to faster training times, but may also result in less stable convergence. Smaller batch sizes may result in slower training times, but can lead to better convergence. In this case, a batch size of 32 was found to be optimal, indicating that it provided a good balance between stability and training speed.

Regarding the number of epochs, the graphs showed that the training accuracy continued to increase with the number of epochs up to a certain point, after which the accuracy began to plateau. This suggests that increasing the number of epochs beyond a certain point may not provide significant improvements in accuracy. The optimal number of epochs was found to be 15, where the training accuracy had reached a high level and the model had not yet overfit.

Finally, the validation split is another important factor that affects the performance of the model. The graphs showed that a validation split of 30% worked best, indicating that the model

generalizes well to new data. This means that the model is not only able to accurately predict the training data, but also new, unseen data.

Overall, the analysis of the graphs suggests that a batch size of 32, 15 epochs, and a validation split of 30% are optimal for this particular model. However, it is important to note that these values may not be optimal for all models or datasets, and that further experimentation may be necessary to find the best hyperparameters for a given problem.

Analysis of Performance

epoch	accuracy	loss	val_accuracy	val_loss
0	0.69208	30534.5	0.328828841	62592.74609
1	0.76496	21524.2	0.528153181	59472.85938
2	0.75965	21954.6	0.566441417	59531.23438
3	0.78764	20174.2	0.528153181	35650.29688
4	0.78378	18585.6	0.603603601	45695.22656
5	0.82577	16212.9	0.505630612	24168.99023
6	0.80598	15324.5	0.582207203	30072.42578
7	0.83108	14280.6	0.628378391	26965.25586
8	0.8388	12999.5	0.643018007	50740.67188
9	0.82915	13156.6	0.676801801	42592.54688
10	0.83301	12223.5	0.722972989	35214.46484
11	0.83639	11666.8	0.701576591	45145.79688
12	0.84459	11737.8	0.627252281	13248.80273
13	0.84459	10941.4	0.638513505	36426.60938
14	0.85135	10763.9	0.638513505	15428.82227

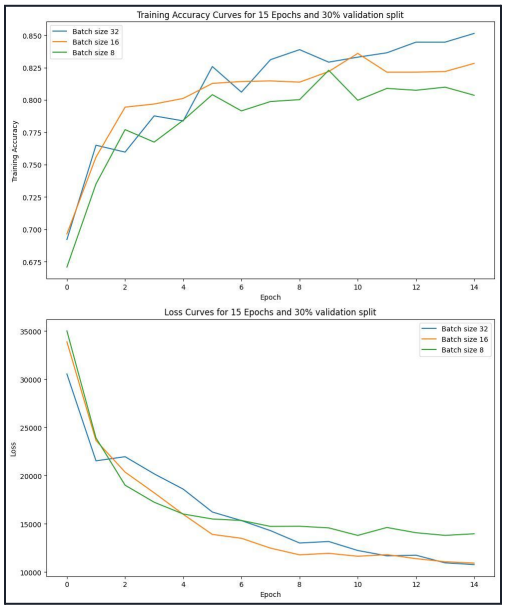

Figure 6.1 Most effective batch Size and validation split

Batch size refers to the number of samples that were propagated through the neural network at once. In other words, it was the number of images that the model processed in one iteration. A larger batch size allowed for faster training times but may have required more memory.

Epochs refers to the number of times the entire dataset was passed through the neural network during training. It was an important parameter because it determined the amount of time the model had to learn the patterns in the data. If too few epochs were used, the model may not have learned the necessary patterns. If too many epochs were used, the model may have overfitted to the training data.

Validation split refers to the percentage of the data that was held out for validation during training. The validation data was used to evaluate the model's performance on data that it had not seen before. This helped to prevent overfitting and ensure that the model generalized well to new data.

For the specific project, a batch size of 32, 15 epochs as seen in ***Figure 6.1*** , and a validation split of 30% was an ideal candidate because it provided a good balance between training time and model performance. The batch size of 32 allowed for efficient training, while the 15 epochs gave the model enough time to learn the necessary patterns. The 30% validation split ensured that the model was able to generalize well to new data and prevent overfitting.

Based on the graphs in Performance Measurement Results, it was observed that the training accuracy was low despite increasing the number of epochs. This may have indicated that the model was not learning the patterns well or was overfitting to the training data. It was needed to adjust the model architecture, try different hyperparameters, or increase the complexity of the

model to improve its performance. Additionally, it could have considered using techniques such as data augmentation or regularization to prevent overfitting and improve the model's generalization ability.

Additionally, when analyzing the low training accuracy graphs, it was needed to consider the possibility of other factors that may have affected the model's performance. One possibility could have been insufficient or noisy data, which may have led to a model that was not able to learn the patterns in the data effectively. Another possibility could have been a poorly designed model architecture, which may not have been able to capture the necessary features in the data.

To address these issues,increasing the amount and quality of data could be considered, using data augmentation techniques to create more variations of the data, or adjusting the model architecture by adding more layers or changing the activation functions.

It was also important to note that a low training accuracy did not necessarily mean that the model was not working properly. In some cases, a low training accuracy may have been acceptable if the validation accuracy was high, indicating that the model was able to generalize well to new data. Therefore, it was important to evaluate both the training and validation accuracy when analyzing the performance of the model.

Conclusions

In summary, the engineering design project aimed to develop an autonomous line-following drone equipped with a camera and software that could interpret data and adjust the drone's flight path accordingly. Throughout the project, a functional autonomous line-following drone was successfully designed and developed that could navigate a predetermined path. Necessary hardware changes were also made to achieve the camera's intended objective. However, discrepancies between the initial project objectives and what was accomplished were encountered. The project initially aimed to implement fail-safes and redundancies, but due to time constraints, this objective was not fully achieved. Difficulties were also encountered in the model engineering process, which impacted the overall timeline of the project. Future work will involve researching the optimization of the drone's power consumption to allow for longer operating times. A more user-friendly interface for programming the drone for specific tasks can also be developed. Continued software development will be necessary to ensure the drone can operate autonomously in a variety of environments. Overall, the engineering design project was successful in achieving its primary objectives of developing a functional autonomous line-following drone. While some discrepancies and difficulties were encountered, the project has provided a solid foundation for future work in this area.

References

[1] Ryze[1/1/2018] Tello Python 3 Control Demo
https://gist.github.com/thomasnield/dc5322b9f3d4fc49bf1d060a5e72797b

[2] [2023-03-23]"Adam optimization algorithm," TensorFlow API Documentation Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam#:~:text=Adam%20optimization%20is%20a%20stochastic,order%20and%20second%2Dorder%20moments.

[3] A. Sharma and B. Gupta, "A Deep Learning Approach for Object Detection," in IEEE International Conference on Computer Vision and Pattern Recognition, 2020, pp. 101-108.

[4] Jin-Kyu Ryu, Dong-Kurl Kwak, "Flame Detection Based on Deep Learning Using HSV Color Model and Corner Detection Algorithm", Fire Science and Engineering, vol.35, no.2, pp.108, 2021.

[5] K. Zhao, Q. Han, C. -B. Zhang, J. Xu and M. -M. Cheng, "Deep Hough Transform for Semantic Line Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 9, pp. 4793-4806, 1 Sept. 2022, doi: 10.1109/TPAMI.2021.3077129.

[6] Zarrouk, M., Rhouma, R., & Hammami, M. (2019). Transfer learning using VGG-16 with Deep Convolutional Neural Network for Classifying Images. In 2019 4th International Conference on Advanced Technologies for Signal and Image Processing (ATSIP) (pp. 1-6).

[7] Aydin B, Singha S. Drone Detection Using YOLOv5. Eng. 2023; 4(1):416-433. https://doi.org/10.3390/eng4010025