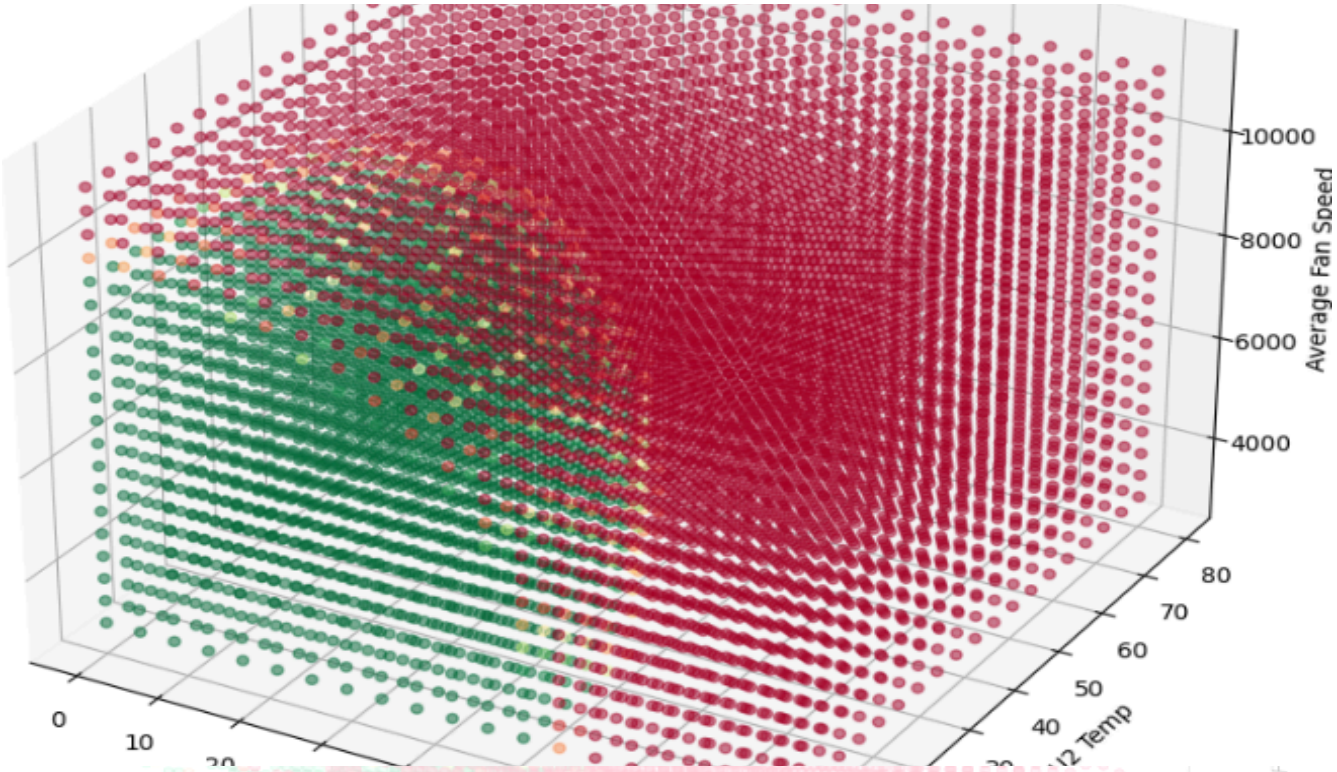


Server Longevity and Efficiency: Transforming Maintenance with AI and TensorFlow

09.04.20XX

O'Neil Stefan Smith

UWLiD: 21500835

Applications of AI

Dr Nasser Matoorian

Abstract

In today's digital landscape, physical servers form the backbone of data centres, IT infrastructures and small businesses, playing a crucial role in ensuring seamless data management, security, and processing (Kumar, U. 2018). As reliance on these servers grows, maintaining their optimal performance becomes vital for business continuity. Traditional maintenance approaches, including reactive and preventive strategies, struggle to anticipate potential failures, leading to unexpected downtime and costly repairs. This project explores the transformative potential of predictive maintenance, harnessing the power of artificial intelligence (AI) and machine learning (ML) for server upkeep. By leveraging TensorFlow, a cutting-edge ML framework, the project aims to design and develop a predictive maintenance model that analyses server sensor data, such as CPU temperature and fan speed, to predict maintenance needs accurately. This predictive approach aims to reduce downtime, extend server lifespan, and optimise operational efficiency. The final model, implemented within a Jupyter notebook, will provide real-time insights into server health, guiding proactive maintenance decisions and revolutionising how server management is approached in enterprise IT infrastructures.

The sensor data in question is not historical, this project will use data collected manually from physical servers discussing how it is collected, organised, manipulated whilst providing justifications as i proceed.

In addition, there will be accompanying youtube videos, made by myself, to further add context to the following chapters: “Servers”, “Data Collection” and “Data Organisation”

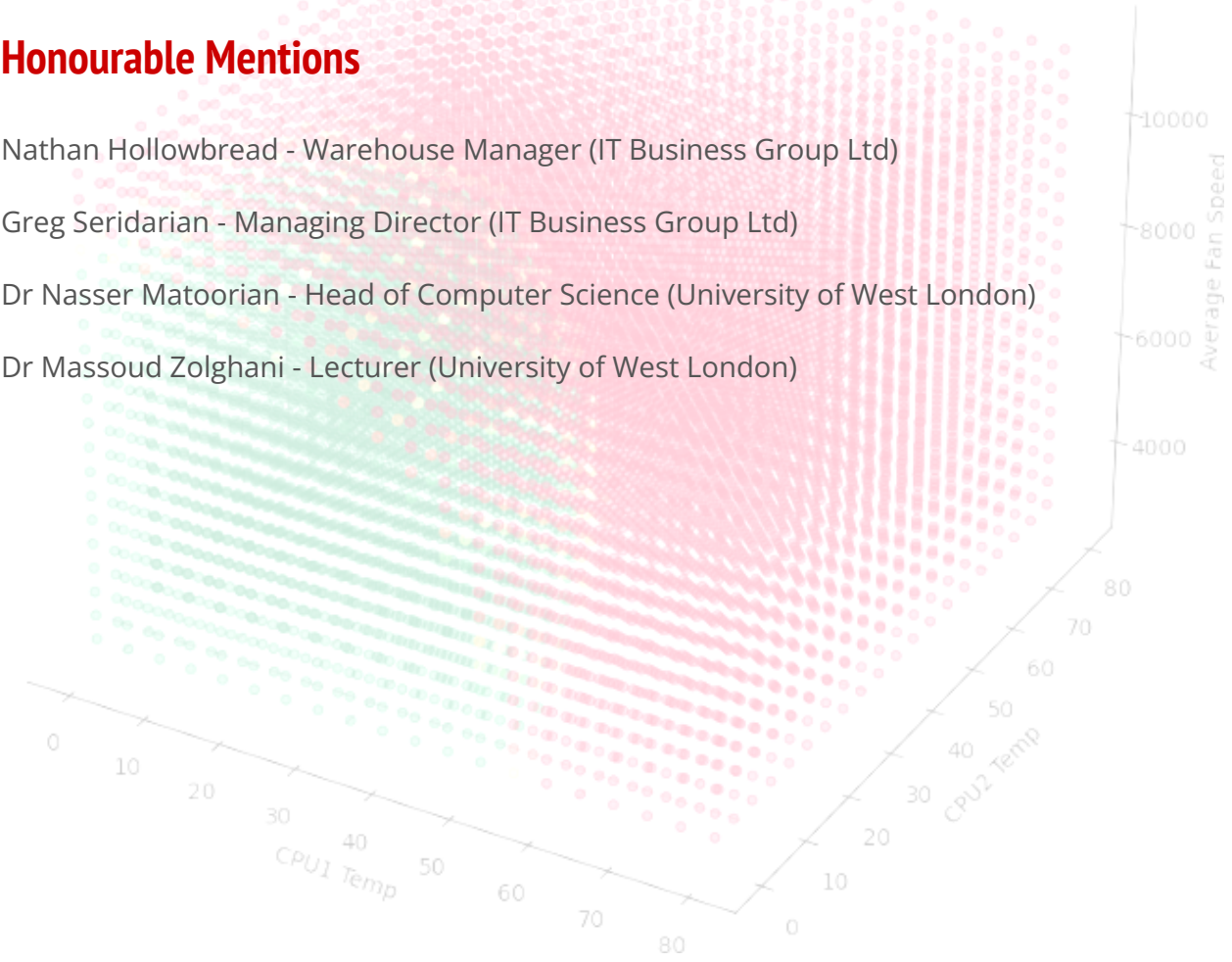
Honourable Mentions

Nathan Hollowbread - Warehouse Manager (IT Business Group Ltd)

Greg Seridarian - Managing Director (IT Business Group Ltd)

Dr Nasser Matoorian - Head of Computer Science (University of West London)

Dr Massoud Zolghani - Lecturer (University of West London)



Background

The advancement of predictive maintenance (PdM) strategies, as part of the transformative wave of Industry 4.0, is deeply intertwined with the rapid development in machine learning (ML) and artificial intelligence (AI). This evolution marks a significant departure from traditional maintenance practices, moving towards more sophisticated, data-driven approaches.

The integration of machine learning and web frameworks into PdM has enabled predictive models to be packaged and deployed with ease, enhancing real-time monitoring capabilities. By utilising tools like TensorFlow, Flask, Docker, and Google Cloud AI Platform, the predictive maintenance landscape is transformed to foster real-time insights and enhanced decision-making capabilities.

The integration of the Internet of Things (IoT) into industrial systems has been a game-changer, facilitating the unification of various devices into a cohesive system. This integration allows for predictive and preventative maintenance strategies to use advanced ML algorithms, fundamentally altering the maintenance landscape in industrial settings (Liulys, 2019). In our project, IoT's role in gathering and analysing vast amounts of data is pivotal in enhancing equipment efficiency and reliability, aligning with Industry 4.0's ethos of interconnected and intelligent systems.

Traditional maintenance models, often relying on the monitoring of setpoints and actual process values, are increasingly being replaced by ML models, especially neural networks. These advanced models offer a nuanced understanding of the dynamic nature of modern industrial systems, learning from data and evolving over time to improve predictive accuracy. Techniques like gradient boosting and neural networks with activation functions like leaky-ReLU have been highlighted for their enhanced predictive capabilities, providing stronger gradients and reducing bias (Liulys, 2019). We've used TensorFlow and Keras to implement a deep learning model that offers sophisticated predictive capabilities, fully aligned with the evolution of PdM strategies.

The transition from simple Run-to-Failure (R2F) methods to more complex and efficient PdM systems is well documented. PdM systems now predict pending failures using

historical data, defined health factors, and statistical inference methods. This shift, driven by increasing data availability and the capabilities of modern hardware and algorithms, underscores the growing effectiveness of ML solutions in maintenance management (Susto et al., 2015). By containerizing the model using Docker, we ensured that our predictive model is portable and easily deployable across different environments.

The research trend in ML has shifted to more complex models, such as ensemble methods and deep learning, due to their higher accuracy in managing large datasets. The rise of deep learning, in particular, owes much to advancements in computing power, notably the evolution of GPUs. These developments have made deep learning a prominent research focus, with its ability to handle the complexities of large-scale industrial data (Research Paper, 2020). In our project, the TensorFlow library was crucial for building and training neural networks capable of deciphering patterns in server data, enabling early detection of potential issues.

Industry 4.0, characterised by cyber-physical systems and the industrial internet of things, integrates software, sensors, and intelligent control units. This integration has enabled automated predictive maintenance functions, analysing massive amounts of process-related data based on condition monitoring (CM). PdM stands out as the most cost-optimal maintenance type, with the potential to achieve an overall equipment effectiveness (OEE) above 90% and promising substantial returns on investment. Maintenance optimization has become a priority for industrial companies, with effective maintenance strategies capable of reducing costs significantly by addressing failures proactively (Research Paper, 2020). By leveraging Google Cloud AI Platform, our project successfully deployed a machine learning model to predict server statuses in real time, offering scalable and reliable deployment.

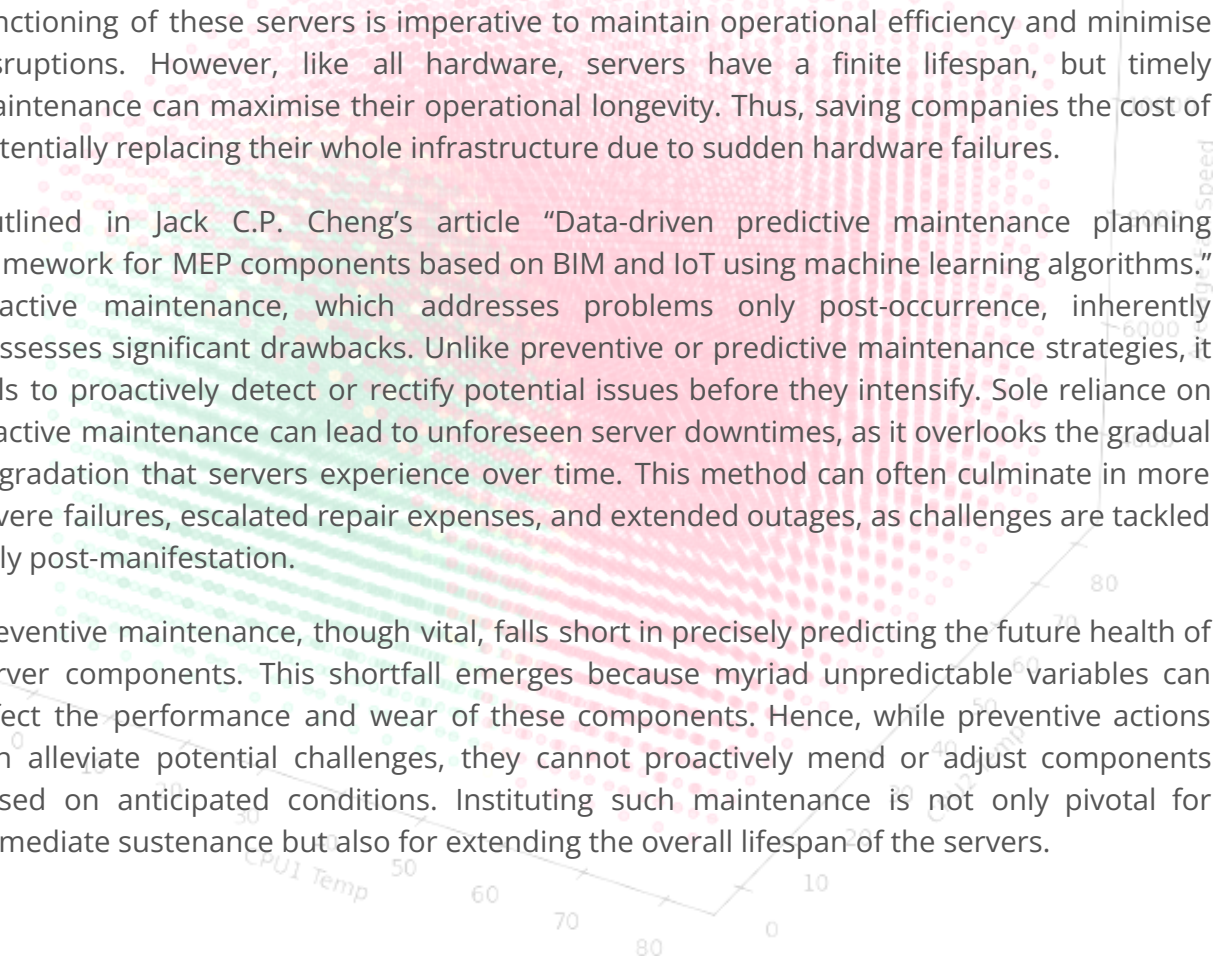
In summary, the literature review underscores the significant advancements in predictive maintenance brought about by the integration of ML and AI technologies. These advancements, particularly in the era of Industry 4.0, have led to a fundamental shift in how maintenance is approached, promising enhanced efficiency, reduced downtime, and overall improved operational efficacy in industrial settings. By using advanced machine learning models, web frameworks, Docker, and Google Cloud, our project showcases the future of predictive maintenance strategies that are increasingly proactive and data-driven.

Introduction

In the modern digital age, physical servers form the foundation of data centres and enterprise IT infrastructures. As businesses have expanded and digitised, the dependence on these robust servers has surged dramatically. These servers not only manage vast amounts of data but also ensure high availability, bolster security, and enhance processing speed. Consequently, the performance and reliability of these servers directly impact the efficiency of IT operations and the overall business continuity. Ensuring the optimal functioning of these servers is imperative to maintain operational efficiency and minimise disruptions. However, like all hardware, servers have a finite lifespan, but timely maintenance can maximise their operational longevity. Thus, saving companies the cost of potentially replacing their whole infrastructure due to sudden hardware failures.

Outlined in Jack C.P. Cheng's article "Data-driven predictive maintenance planning framework for MEP components based on BIM and IoT using machine learning algorithms." Reactive maintenance, which addresses problems only post-occurrence, inherently possesses significant drawbacks. Unlike preventive or predictive maintenance strategies, it fails to proactively detect or rectify potential issues before they intensify. Sole reliance on reactive maintenance can lead to unforeseen server downtimes, as it overlooks the gradual degradation that servers experience over time. This method can often culminate in more severe failures, escalated repair expenses, and extended outages, as challenges are tackled only post-manifestation.

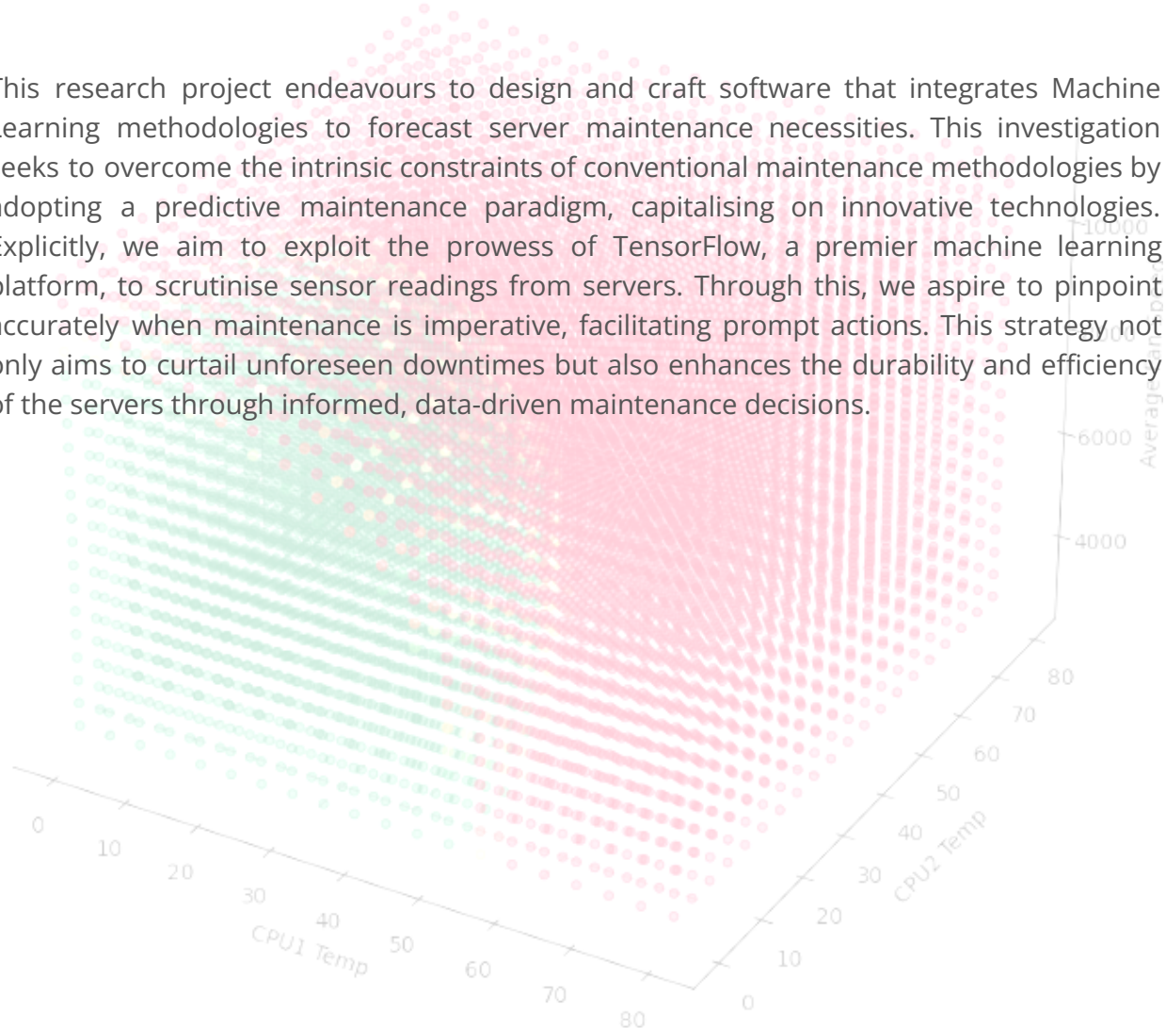
Preventive maintenance, though vital, falls short in precisely predicting the future health of server components. This shortfall emerges because myriad unpredictable variables can affect the performance and wear of these components. Hence, while preventive actions can alleviate potential challenges, they cannot proactively mend or adjust components based on anticipated conditions. Instituting such maintenance is not only pivotal for immediate sustenance but also for extending the overall lifespan of the servers.



Predictive maintenance, underpinned by Artificial Intelligence (AI), stands as a revolutionary advancement in the realm of server upkeep. Shown by Tyagi, V. et al. (2020), Unlike its reactive and preventive counterparts, predictive maintenance delves deep into historical and real-time data, employing sophisticated AI algorithms to discern patterns and anomalies that might be imperceptible to human analysis. By continuously monitoring server health and analysing vast datasets, AI-driven predictive maintenance can forecast potential issues long before they manifest. This not only allows IT teams to intervene

proactively, minimising disruptions, but also tailors' maintenance schedules based on actual server conditions rather than generic timelines. The integration of AI transforms the maintenance paradigm from a schedule-driven approach to a data-driven one, ensuring that servers operate at their peak efficiency while significantly reducing unforeseen downtimes. In essence, AI-augmented predictive maintenance heralds a future where server malfunctions are anticipated and mitigated, ensuring seamless and optimised IT operations.

This research project endeavours to design and craft software that integrates Machine Learning methodologies to forecast server maintenance necessities. This investigation seeks to overcome the intrinsic constraints of conventional maintenance methodologies by adopting a predictive maintenance paradigm, capitalising on innovative technologies. Explicitly, we aim to exploit the prowess of TensorFlow, a premier machine learning platform, to scrutinise sensor readings from servers. Through this, we aspire to pinpoint accurately when maintenance is imperative, facilitating prompt actions. This strategy not only aims to curtail unforeseen downtimes but also enhances the durability and efficiency of the servers through informed, data-driven maintenance decisions.



Aims and Objectives

Aim:

The primary aim of this project is to develop a predictive maintenance software solution using machine learning (ML) that can accurately predict server maintenance needs based on self attained and real-time sensor data. This project seeks to harness the power of TensorFlow to build a predictive model that can analyse key server metrics such as CPU temperature, fan speed, and error logs to forecast potential failures. The goal is to transform maintenance from a reactive or scheduled process to one driven by data, enabling businesses to optimise server longevity and operational efficiency.

Objectives:

Data Collection and Preprocessing:

Objective: To gather comprehensive sensor data from enterprise servers and preprocess it for analysis.

Justification: The quality of the predictive model depends heavily on the input data. Accurate and relevant data ensures that the model captures the patterns and anomalies necessary to predict maintenance needs effectively.

Exploratory Data Analysis:

Objective: To identify patterns and trends within the collected data that correlate with server health and failures.

Justification: Understanding the relationships within the data provides insight into which features are most predictive of failures, guiding the feature engineering and model design processes.

Model Design and Development:

Objective: To design a predictive maintenance model using TensorFlow, focusing on binary classification to determine if a server needs maintenance.

Justification: Binary classification allows for a straightforward assessment of server health, simplifying the interpretation of results and enabling quick decision-making to prevent downtime.

Model Training and Validation:

Objective: To train the predictive model using the collected data and validate its performance through testing.

Justification: Training the model on real-world data ensures it learns patterns specific to the target environment, while validation confirms the model's accuracy and ability to generalise to unseen data.

Integration and Deployment:

Objective: To deploy the predictive model in a real-time environment, incorporating it into an AI platform for live server monitoring.

Justification: Integrating the model into a real-time monitoring system allows IT teams to receive predictive maintenance alerts, enabling them to act proactively to minimise disruptions and optimise server efficiency.

Analysis of Practical Impact:

Objective: To evaluate the impact of the predictive maintenance model on server uptime, maintenance costs, and operational efficiency.

Justification: Analysing the potential and practical benefits of the model will demonstrate its value in real-world applications, supporting broader adoption of predictive maintenance strategies in the industry.

Introduction into Essential ML Tools

In the realm of Machine Learning (ML), several tools and libraries have become standard for professionals and enthusiasts alike due to their powerful features and ease of use. Here's an overview of some essential tools, each pivotal in the ML workflow:

TensorFlow

TensorFlow is an end-to-end open-source platform designed for machine learning. Developed by the Google Brain team, it has grown to be one of the most widely used ML libraries in the industry. TensorFlow excels in providing a comprehensive toolkit for researchers and developers to develop advanced ML models. One of its standout features is the ability to build and train neural networks to detect and decipher patterns and correlations, analogous to learning and reasoning used by humans. It supports a range of tasks from regression, classification, and prediction, all the way to more complex functions like natural language processing and image recognition.

TensorFlow's architecture allows for deployment across a variety of platforms (CPUs, GPUs, and even TPUs). It offers multiple abstraction levels for choosing the right one for your needs – from direct TensorFlow API commands that allow for intricate operation control to high-level Keras API which facilitates common model design patterns with ease. The flexibility and scalability of TensorFlow make it suitable not just for research and development but also for production deployment.

NumPy

NumPy is the foundational package for scientific computing in Python. It offers a powerful N-dimensional array object, sophisticated functions, tools for integrating C/C++ and Fortran code, and useful linear algebra, Fourier transform, and random number capabilities. For machine learning practitioners, NumPy is indispensable for data manipulation and preprocessing. It enables numerical operations on large data sets with speed and efficiency that native Python data structures cannot match, due to its underlying C-optimised code.

NumPy arrays form the backbone of nearly all data structures used in machine learning models, providing a much more efficient way to store and manipulate data than traditional

Python lists. By facilitating operations on large arrays and matrices, NumPy serves as the bedrock upon which other libraries, including pandas and scikit-learn, are built.

Pandas

Pandas is a critical tool in the data scientist's toolkit, designed to work with structured data intuitively. It is particularly well-suited for data manipulation and analysis, offering data structures like DataFrame and Series, which are not only easy to use but also powerful for handling real-world data. pandas support a variety of data formats, allowing for easy data import, export, and manipulation.

With pandas, data scientists can perform tasks ranging from data cleaning and transformation to more sophisticated operations like data aggregation and time-series analysis. Its merging and joining capabilities are especially useful for combining datasets in complex ways, facilitating more in-depth analysis and modelling.

Matplotlib

Matplotlib is a versatile visualisation library in Python, capable of producing a wide range of static, animated, and interactive visualisations. In the context of machine learning, it is invaluable for exploratory data analysis, allowing practitioners to visualise trends, patterns, and outliers in the dataset. Through plots like histograms, scatter plots, and line charts, Matplotlib helps in understanding the data's underlying distribution, correlations, and structure.

Effective visualisation is crucial not only for exploratory analysis but also for communicating results and findings. Matplotlib provides a highly customizable interface for creating publication-quality figures and graphics that can convey complex data insights in a comprehensible and visually appealing format.

Scikit-learn

Scikit-learn is a premier library providing efficient tools for machine learning and statistical modelling including classification, regression, clustering, and dimensionality reduction. Built on NumPy, SciPy, and Matplotlib, scikit-learn offers an accessible yet versatile framework for data mining and data analysis.

Its appeal lies in its easy-to-use API and comprehensive documentation that guides users through the various algorithms it supports. With functions for fitting models, data preprocessing, cross-validation, and many more, it is designed to interoperate seamlessly with NumPy and pandas, making it a linchpin in the Python data science stack. Its consistent interface across different types of algorithms simplifies the process of experimenting with and deploying various models, making it an ideal toolkit for both novice data scientists and seasoned practitioners alike.

Flask

Flask is a micro web framework written in Python that is lightweight and easy to use. In machine learning, Flask is often used to develop web-based APIs that can serve model predictions. Its simplicity and flexibility make it suitable for developing rapid prototypes and serving lightweight APIs for machine learning models. Flask enables machine learning practitioners to deploy their models quickly as RESTful APIs, providing easy integration with other systems.

Docker

Docker is an open platform for developing, shipping, and running applications in containers. For machine learning projects, Docker allows you to package models, scripts, and dependencies into a single portable container that can run on any platform. This ensures consistency across different environments and simplifies deployment, making it easier to share and collaborate on ML projects.

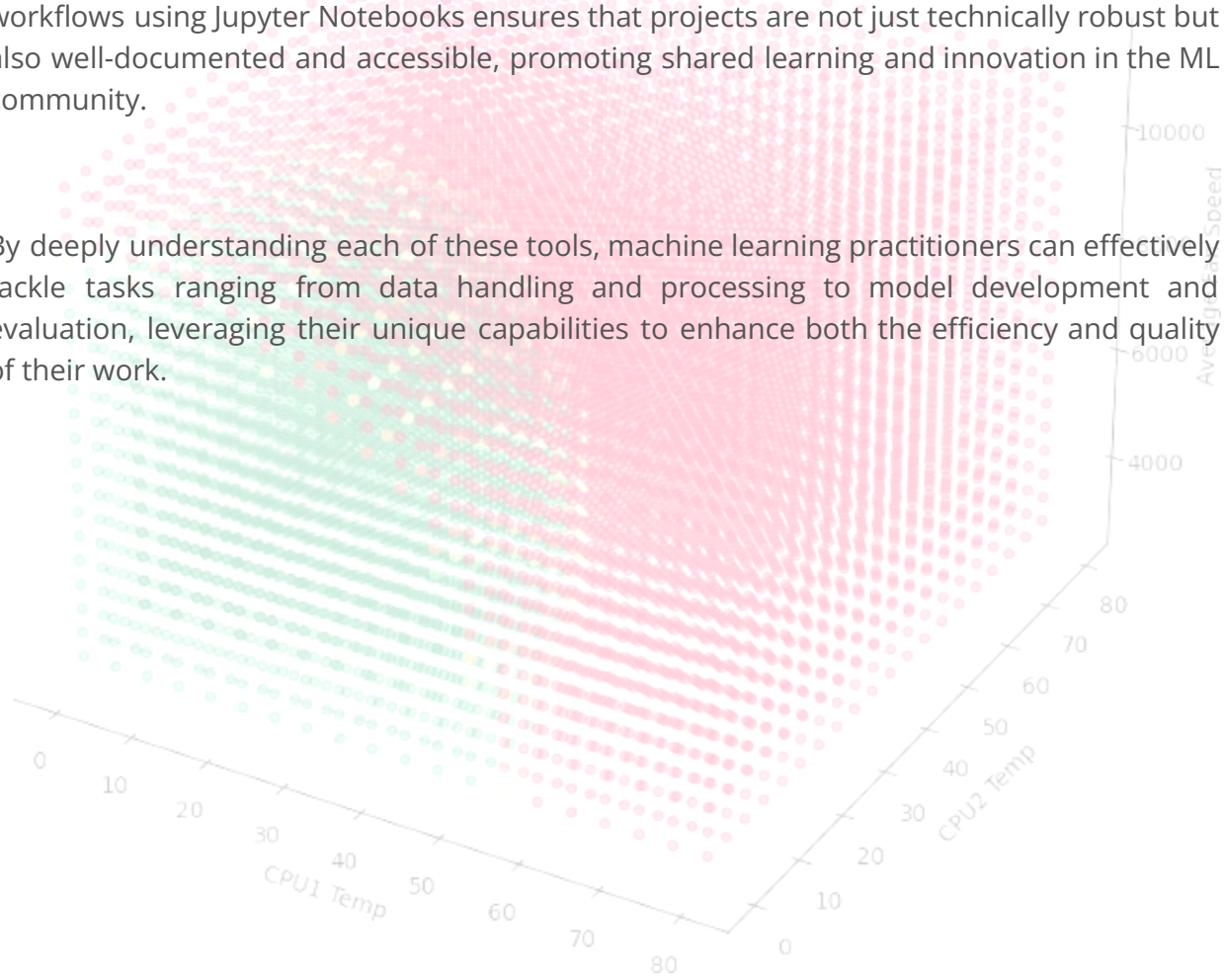
Google Cloud AI Platform

Google Cloud AI Platform is a suite of cloud services designed for deploying and managing machine learning models at scale. By integrating with Google Container Registry, it allows machine learning practitioners to deploy models in containers. This platform supports the entire ML workflow, from data preparation to model serving, and provides a scalable infrastructure to run prediction services.

GitHub

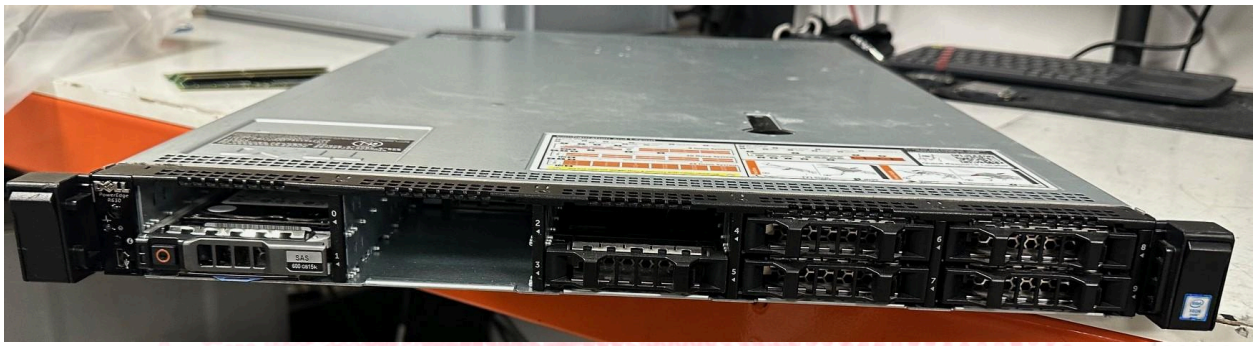
GitHub is a critical platform for version control and collaboration in machine learning projects, offering seamless integration with Jupyter Notebooks to host, manage, and share code and data. This combination enhances project transparency, reproducibility, and teamwork by centralising workflows in a single repository where changes are tracked, facilitating easy revisions and collaboration without data loss or overwrite concerns. GitHub not only streamlines code and project management through features like pull requests and issue tracking but also fosters an open-source community where data scientists and developers share knowledge and advancements. Incorporating GitHub in machine learning workflows using Jupyter Notebooks ensures that projects are not just technically robust but also well-documented and accessible, promoting shared learning and innovation in the ML community.

By deeply understanding each of these tools, machine learning practitioners can effectively tackle tasks ranging from data handling and processing to model development and evaluation, leveraging their unique capabilities to enhance both the efficiency and quality of their work.

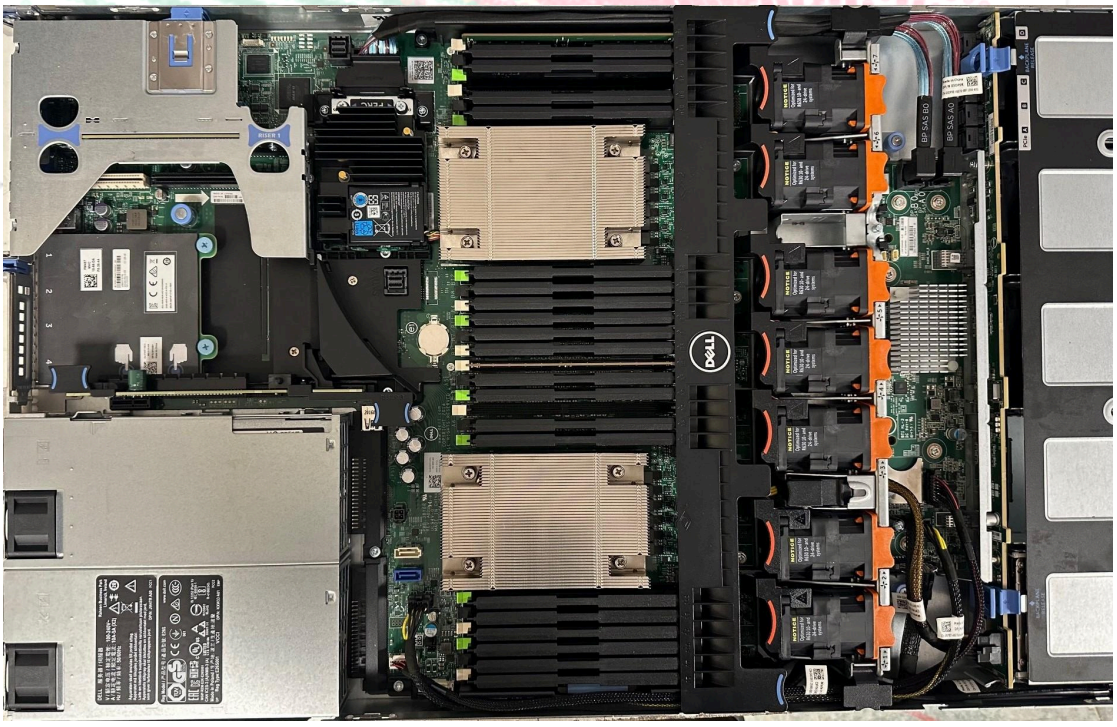


Servers

Video: https://youtu.be/t-vj5j1CkvU



Servers are foundational to the modern world of Information Technology (IT) and beyond, playing a critical role in data processing, storage, and management across various sectors including finance, healthcare, education, and e-commerce. They are the backbone of the internet, hosting the applications and services we use daily, from cloud computing and online banking to streaming services and social media platforms. Servers ensure that data is accessible, secure, and efficiently managed, enabling businesses and organisations to operate seamlessly, analyse big data for insights, and provide the digital services that have become integral to our daily lives.



In the context of maintaining these vital systems, the use of sensors within servers becomes critical. Sensors monitor various parameters, such as CPU temperature and fan speed, providing real-time data that is essential for maintaining operational efficiency and preventing downtime. For instance, overheating can significantly impair a server's performance and, in severe cases, lead to hardware damage. By monitoring CPU temperature and fan speeds, IT professionals can intervene early to mitigate risks, such as by improving ventilation or performing maintenance tasks.

The integration of TensorFlow-based AI algorithms into predictive maintenance methodologies represents a significant advancement in optimising the operational longevity and efficiency of physical servers in data centres and enterprise IT infrastructures. TensorFlow's ability to analyse complex data sets enables the development of models that can predict potential failures or identify inefficiencies in server operations before they become critical issues. For example, by analysing trends in temperature data and fan speed, TensorFlow can predict when a server is likely to overheat or when a fan is failing, allowing for pre-emptive maintenance actions that can avoid costly downtime and extend the server's lifespan.

Furthermore, TensorFlow's machine learning capabilities can optimise workload distribution based on the thermal behaviour of individual CPUs within a server. This ensures that no single CPU is overburdened, reducing the risk of overheating, and improving overall system efficiency. By leveraging such AI-driven insights, Data Centres can significantly enhance their predictive maintenance strategies, leading to more reliable, efficient, and cost-effective operations.

The criticality of servers in IT and the importance of sensor data for their maintenance underscores the value of integrating advanced AI algorithms like those offered by TensorFlow. This integration not only enhances the ability to maintain and optimise server operations but also represents a forward-looking approach to managing the increasingly complex and crucial IT infrastructures that support our digital world.

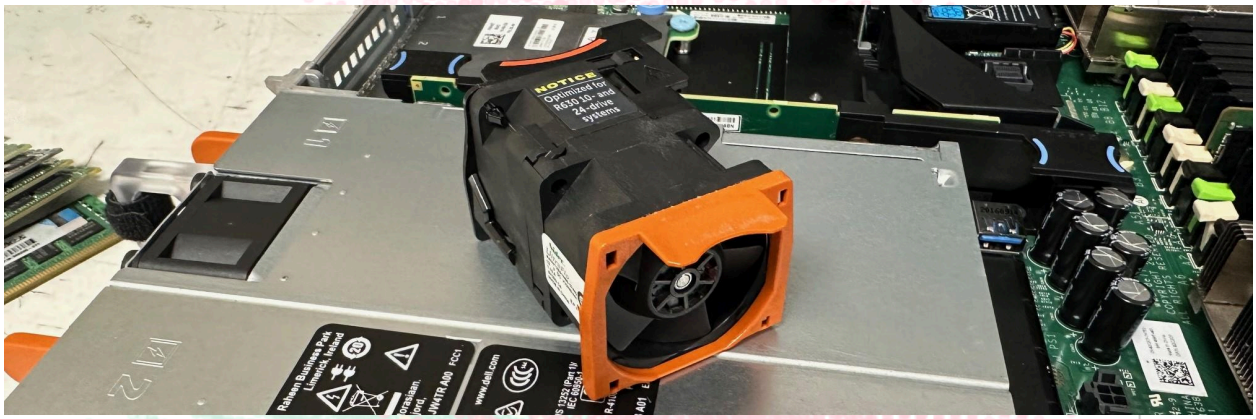
Central Processing Unit (CPU)

The CPU is essentially the brain of the computer, handling millions of processes per second. High temperatures can degrade its performance over time or cause immediate throttling, where the CPU reduces its speed to prevent overheating. This throttling can lead to slower system performance and, in severe cases, system crashes or hardware damage. Monitoring CPU temperature helps in early detection of potential overheating issues, allowing for timely intervention like cleaning dust from the system, improving ventilation, or replacing the thermal paste.

Monitoring each individual CPU's temperature within a server, rather than relying on an average temperature, provides more granular insight into the thermal status of each processing unit, enabling precise identification of localised overheating issues that an average temperature might mask. This detailed monitoring is crucial because even if one CPU overheats while others remain cool, the average temperature could appear normal, potentially overlooking critical hotspots that can lead to CPU throttling or failure. Individual temperature readings allow for targeted cooling adjustments and more effective troubleshooting, ensuring that all CPUs operate within their thermal thresholds for optimal performance and reliability. Additionally, understanding the specific thermal behaviour of each CPU can inform better workload distribution and cooling strategies, enhancing the overall efficiency and longevity of the server.

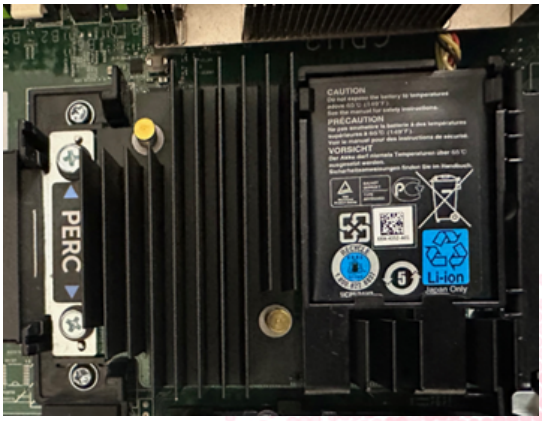
Fans

Fans play a crucial role in maintaining optimal operating temperatures by dissipating heat away from critical components like the CPU and GPU. The fan speed adjusts according to the system's cooling needs; higher temperatures typically demand higher fan speeds. Monitoring fan speeds can provide insights into the overall thermal status of the system. Unusually high fan speeds might indicate excessive heat or inefficient cooling, while unusually low speeds could signal fan malfunctions or obstructions that impede airflow.



Using the average fan speed of a server, as opposed to monitoring its 7 individual fan speeds, offers a more streamlined and efficient approach to understanding the system's thermal health. By focusing on the average speed, administrators can simplify monitoring, reduce data noise, ensure consistent performance evaluation, facilitate predictive maintenance, optimise computational resources, and gain a holistic view of the server's cooling efficiency. This approach not only makes it easier to detect and address potential issues proactively, but also provides a reliable metric that reflects the overall effectiveness of the server's cooling system, ensuring optimal performance and longevity of the hardware components.

RAID Controller (Redundant Array of Independent Disks)



The Redundant Array of Independent Disks (RAID) card is crucial for data redundancy and performance enhancement. It allows multiple hard drives to work together, improving the overall system's fault tolerance and data integrity. In a data collection environment, where the loss or corruption of data can have significant consequences, a RAID setup ensures that data is mirrored across multiple drives. This means that if one drive fails, the system can continue to operate without data loss, and the failed drive can be replaced without downtime. Additionally, RAID can be configured to enhance the read/write speed, essential for the high-speed data transactions typical in server operations.

RAM (Random Access Memory)

Random Access Memory (RAM) is vital for the temporary storage of data that the server's processor needs to access quickly. High-capacity, high-speed RAM is essential for efficient data processing, allowing for faster retrieval and manipulation of data. This is particularly important in data collection scenarios where the server must handle large datasets or run multiple applications simultaneously. Sufficient RAM ensures that these operations can be performed smoothly, without lag or bottlenecks, significantly affecting the server's ability to collect, process, and analyse data efficiently.

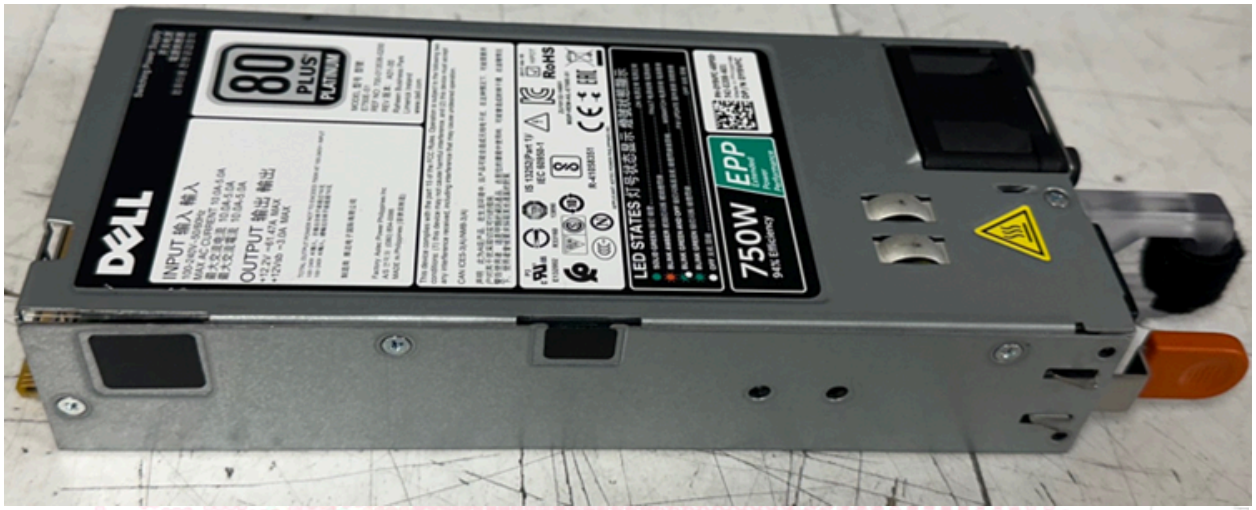
Network Interface Controller

A NIC is fundamental for establishing and managing the server's connection to a network. In the context of data collection, a high-performance NIC ensures that the server can handle high volumes of data ingress and egress without network bottlenecks. It is responsible for the fast and reliable transmission of data between the server and other networked devices or internet-based resources, making it a key component for servers that rely on network-intensive applications or need to transmit collected data to remote storage or analysis services.

Hard Disk Drive

The hard drive serves as the primary storage device for the server, holding the operating system, applications, and, most importantly, the collected data. In a data collection environment, the choice of hard drive impacts the server's storage capacity, speed, and reliability. Solid-state drives (SSDs) offer faster data access speeds than traditional hard disk drives (HDDs), making them preferable for situations where speed is crucial. However, HDDs may still be used for long-term storage of large volumes of data where speed is less critical. The selection between SSDs and HDDs (or a combination of both) depends on the specific needs of the data collection task, including considerations of speed, capacity, and cost.

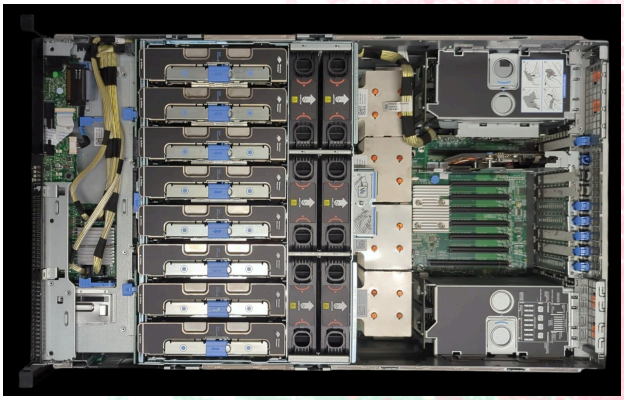
PSU (Power Supply Unit)

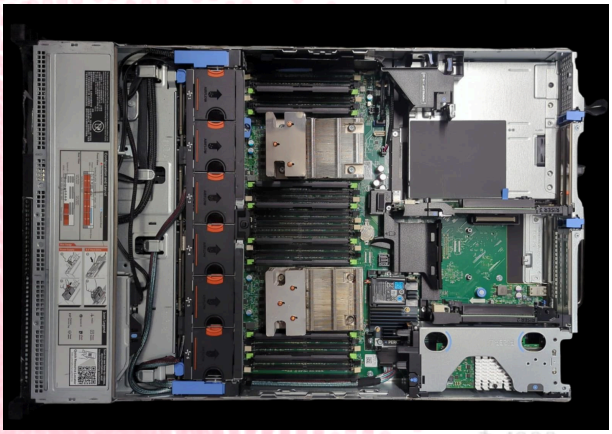


The Power Supply Unit (PSU) is critical for providing stable and reliable power to all the server's components. A high-quality PSU is vital for ensuring the server operates efficiently and without interruption, particularly during intensive data collection and processing tasks. It protects the server against power surges and fluctuations that could lead to data corruption or loss.

Overall

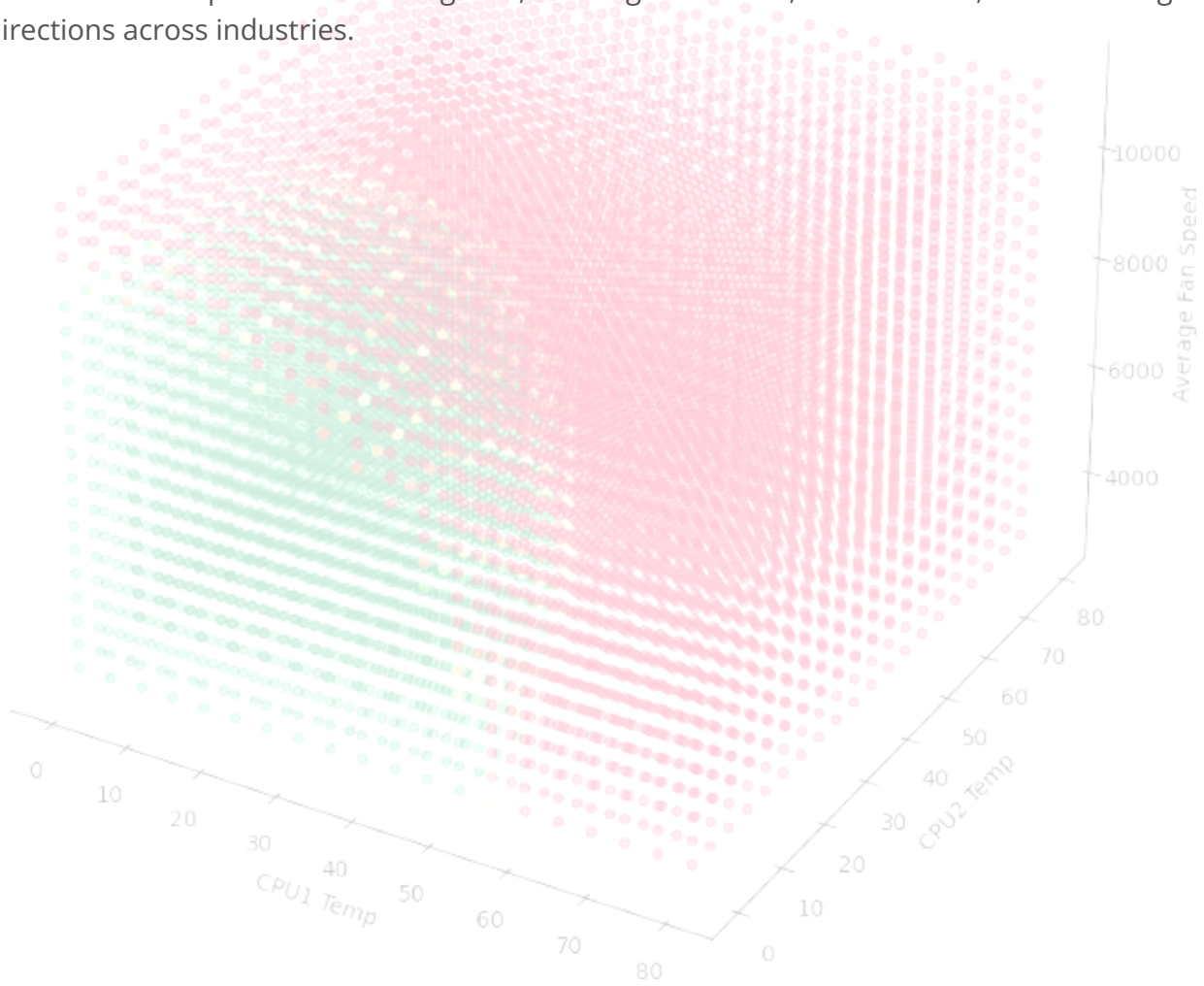
This study, focusing on the use of TensorFlow-based AI algorithms for predictive maintenance, underscores the immense potential of integrating advanced computational techniques within server environments. This integration is crucial not just for servers like the Dell PowerEdge R630, which is used for the sensor data collection of this study, but also across various models employed by different IT companies such as the diverse configurations of the Dell PowerEdge R930, HP ProLiant Gen9 DL580, and Dell PowerEdge R730





Moreover, the comprehensive justification for data collection within server environments, as detailed through the exploration of critical components such as the RAID card, RAM, NIC, PSU, and hard drives, underscores the intricate balance between hardware reliability, efficiency, and the advanced computational requirements of modern IT infrastructures. Each component, from ensuring data redundancy and enhancing processing speed to facilitating robust network connections and providing dependable storage solutions, plays a pivotal role in the overarching goal of optimising server operations for the demanding tasks of data collection and analysis.

This exploration not only highlights the necessity of each hardware component in maintaining the operational integrity and performance of servers but also illuminates the potential for integration with cutting-edge technologies like TensorFlow-based AI algorithms. Such integration promises to revolutionise predictive maintenance methodologies, further enhancing the resilience, efficiency, and longevity of servers in data centres and enterprise IT environments. By leveraging these technologies and insights, organisations can anticipate and preemptively address potential issues, ensuring the continuous, reliable operation of their servers. This proactive approach to server maintenance and optimization is essential in an era where data is not just an asset but the backbone of operational intelligence, driving decisions, innovations, and strategic directions across industries.

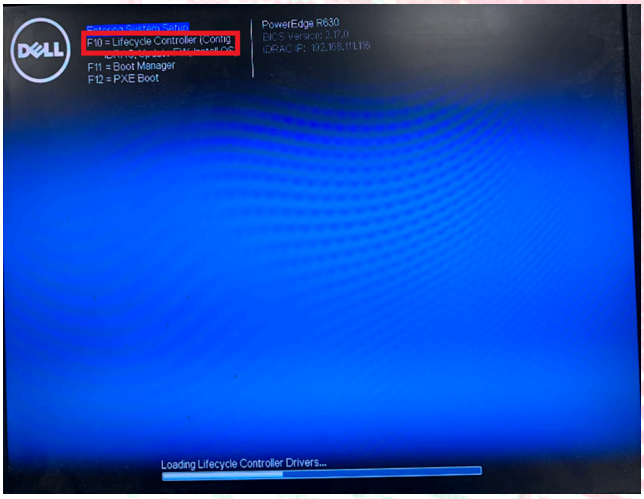


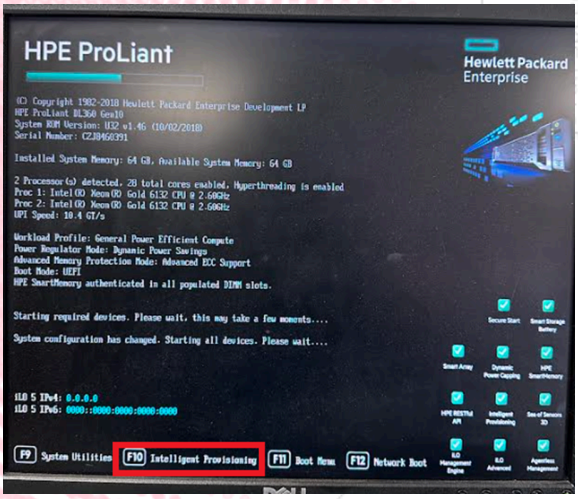
Data Collection

Video: https://youtu.be/bEuZfevDopg

For this particular case, we will be collecting the data (CPU Temperature and fan speed) from the onboard Dell application known as the 'Lifecycle Controller' which is delivered as part of integrated Dell Remote Access Controller (iDRAC) out-of-band solution and embedded Unified Extensible Firmware Interface (UEFI) applications in the latest Dell servers.

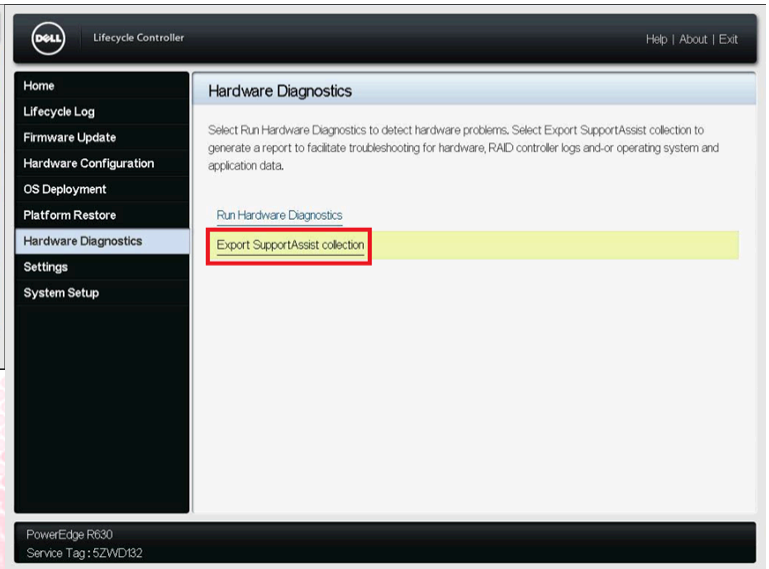
When booting up a 13th Generation Dell Server and the USB Ports have initialised, you will be presented with 4 options. For the purpose of data collection, I will only mention the option that is of concern which is the F10 Option





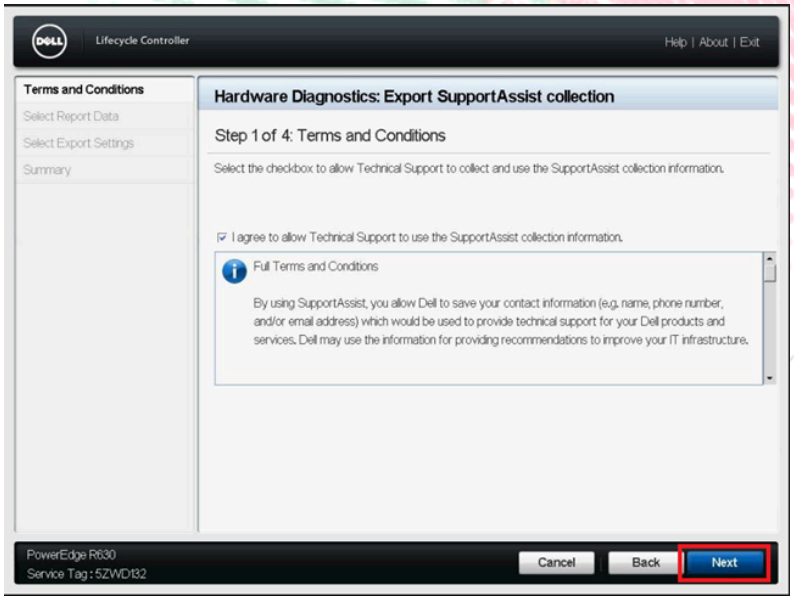
Before we continue any further, it must be mentioned that what is to be shown can also be done with many other brands of server such as IBM and HP with their own and respective management firmware, for instance, the HP Model variant would still list you with 4 options whilst booting and again you will enter F10 to enter its 'Intelligent Provisioning' firmware to then view and collect its sensor data

Going back to the Dell R630 we were looking at initially, after it has booted successfully into the Lifecycle Controller, we will be presented with a host of utilities to enter and in this case, we shall enter the 'Hardware Diagnostics' tab which will allow us to do one of two things:

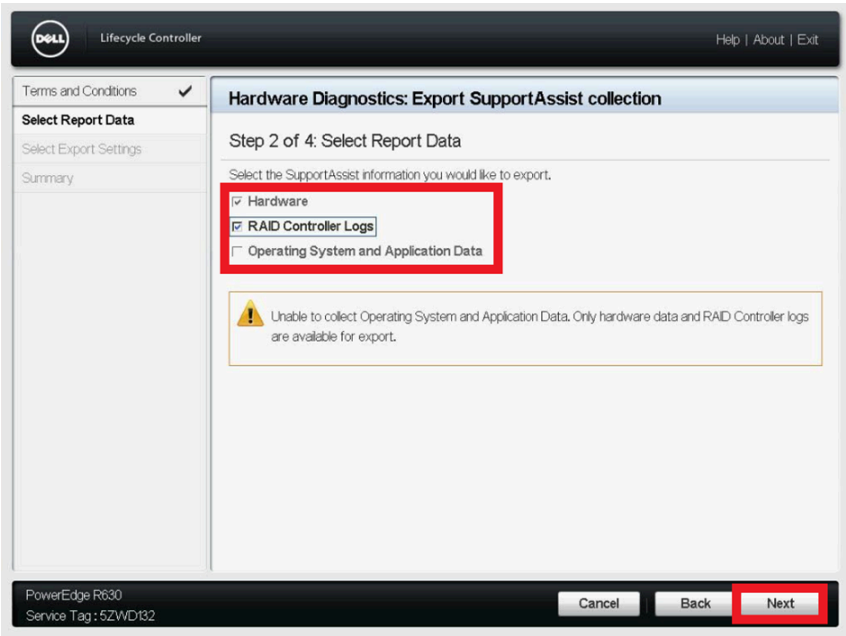


'Run Hardware Diagnostics' or 'Export SupportAssist collection'.

The 1st option will lead us to the preventative/reactive form of maintenance. This test can vary from taking between 20 minutes to 2 hours depending on the issue which can mean a long and undesirable downtime of a system but it does give a detailed breakdown of all tests and results when the server is put under stress or not. We want to export the initial/running diagnosis of all parts of the server where a sensor is located which not only gives us valuable information but also does not take a long time to do (usually around 4 minutes) and with this data, we will use it to build the predictive form of maintenance with the use of TensorFlow and AI.

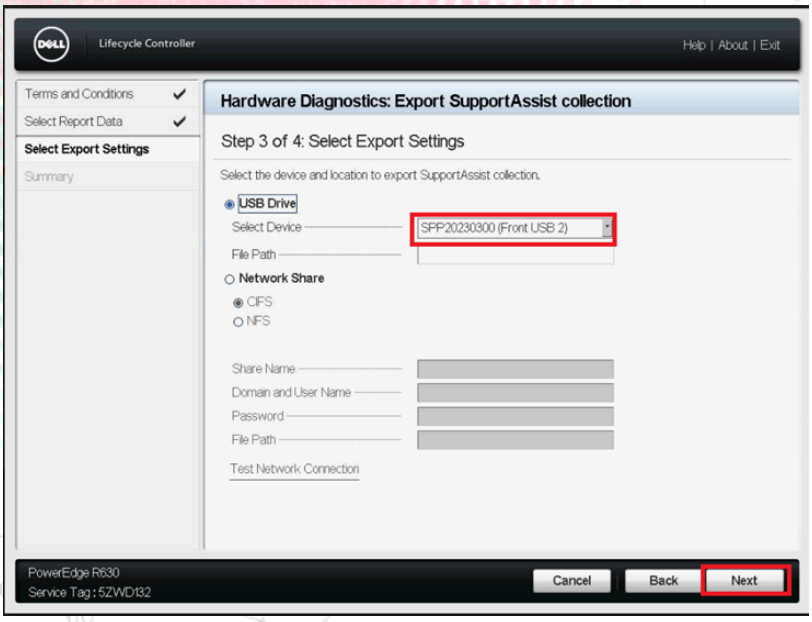


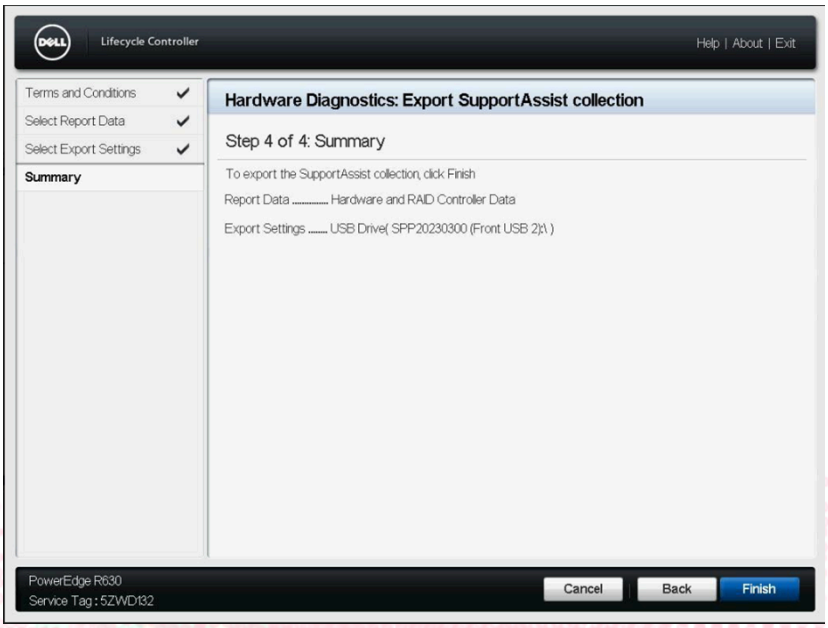
Click "I Agree" to continue ->



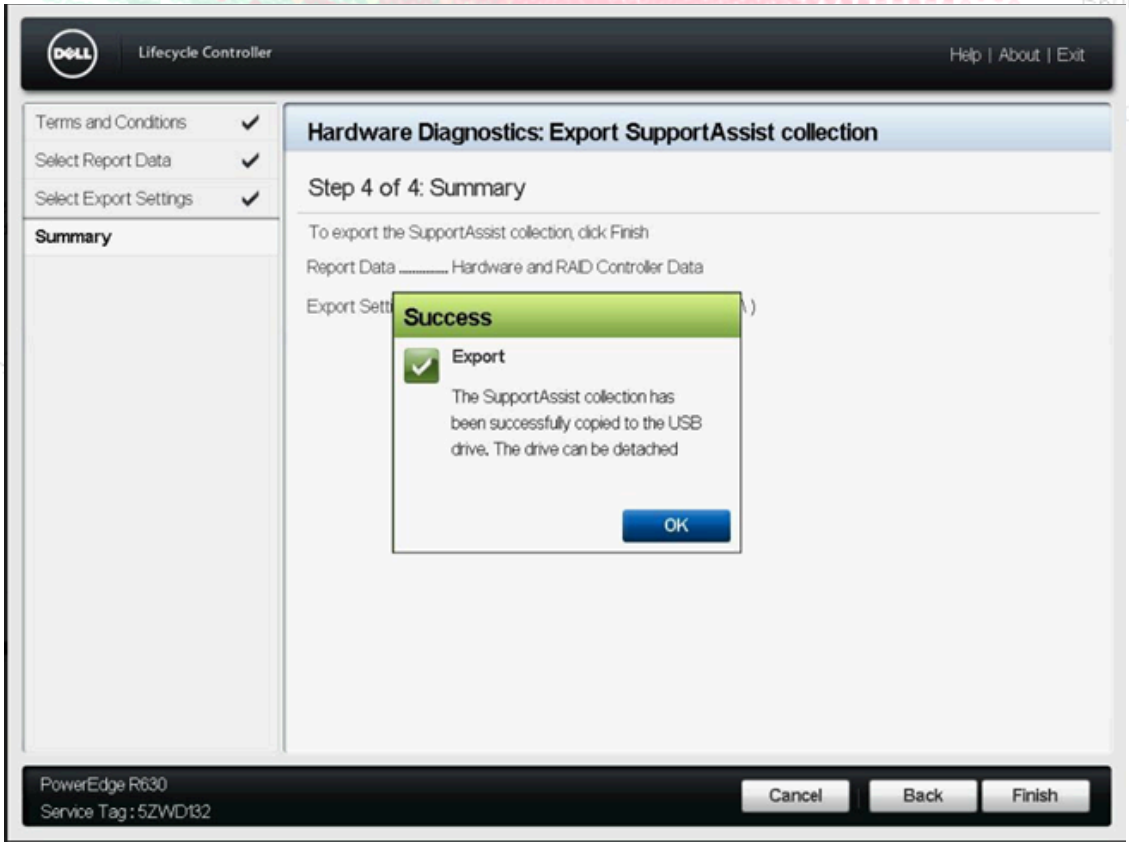
All boxes that can be checked must be checked to ensure we collect the maximum amount of sensor data possible then click next to continue again. ->

This screen dictates where the gathered information should be stored, I chose using a USB for simplicity but via the NFC or CIFS protocol, the information can be directly stored on your computer, or any other virtual drive located.





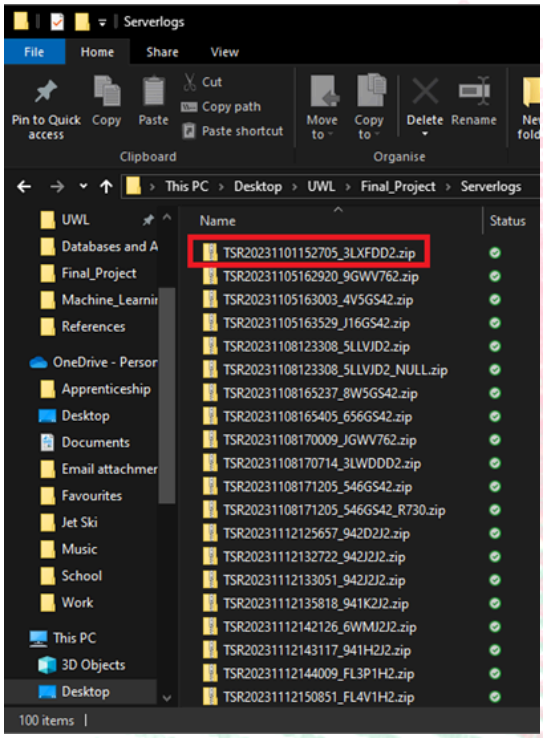
Click finish on the final confirmation screen to then complete this process for 1 server, this whole process must be completed for each server (Data Point)

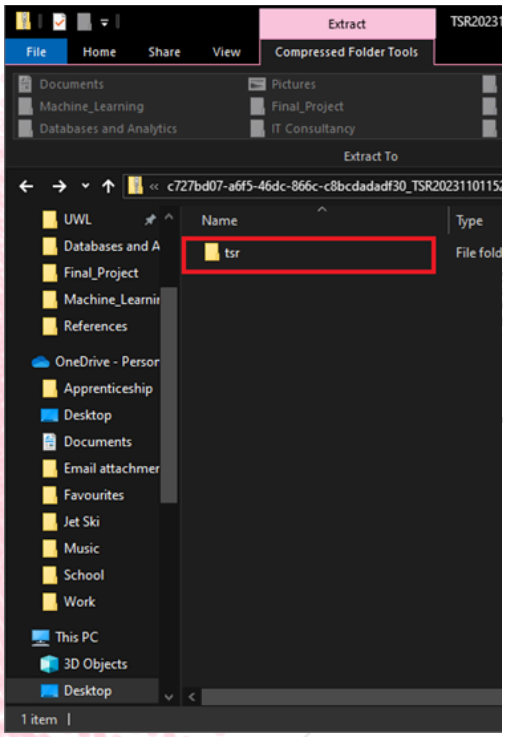


Data Organisation

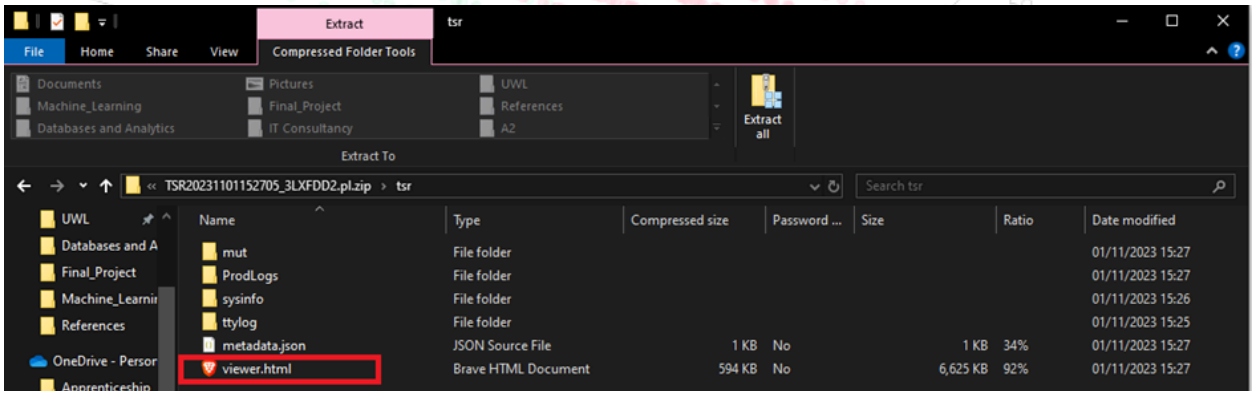
Video: https://youtu.be/yHOxb2ICOXU

After data has been exported from the physical server, it appears in the form of a zip file that when extracted, provides a html viewer of all collected information

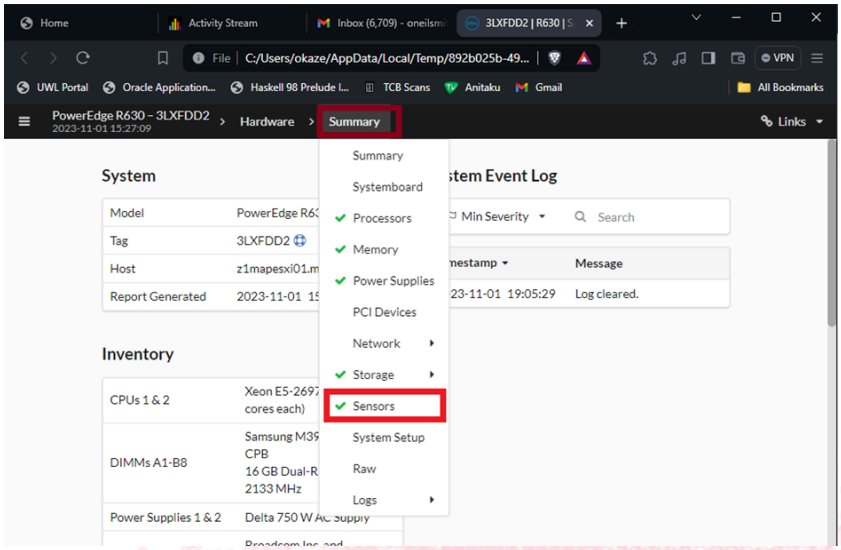




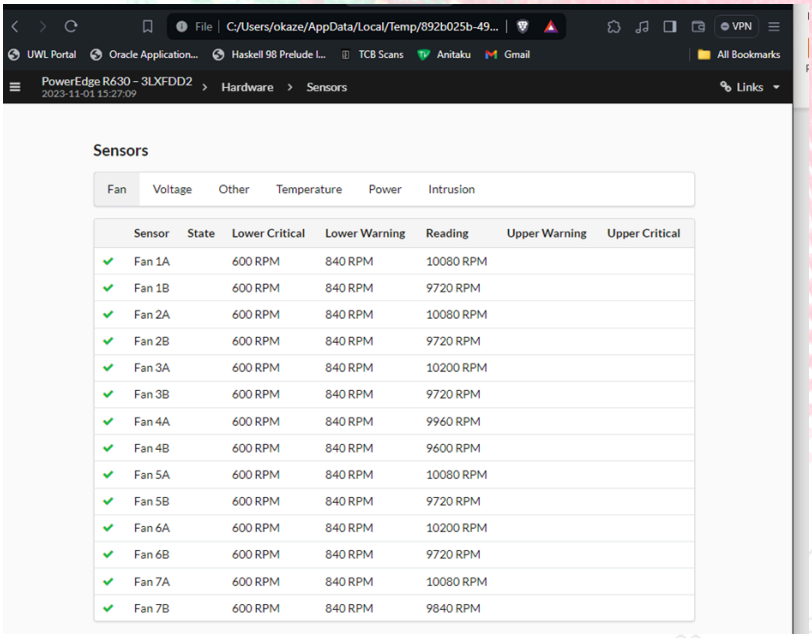
Once extracted you can now begin to view all information provided by the server.



Clicking on “viewer.html” will then take you to the next screen which contains all the data it a neatly contained fashion



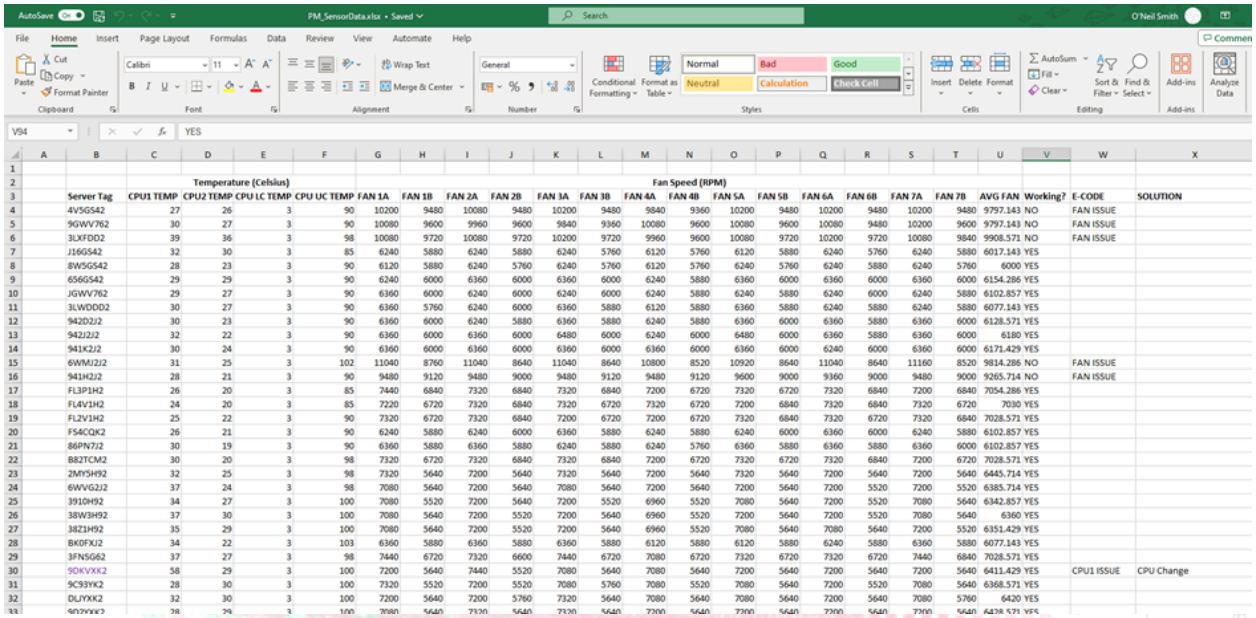
Once arrived at this screen, it can be clearly seen that a whole host of information from the server has been made available to be viewed, from what OS the server is running to which hardware is keeping the server operational. But what we want to see is the data returned by the sensors embedded within the server hence we click on sensors after the summary tab has been clicked



Sensor	State	Lower Critical	Lower Warning	Reading	Upper Warning	Upper Critical
Fan 1A		600 RPM	840 RPM	10080 RPM		
Fan 1B		600 RPM	840 RPM	9720 RPM		
Fan 2A		600 RPM	840 RPM	10080 RPM		
Fan 2B		600 RPM	840 RPM	9720 RPM		
Fan 3A		600 RPM	840 RPM	10200 RPM		
Fan 3B		600 RPM	840 RPM	9720 RPM		
Fan 4A		600 RPM	840 RPM	9960 RPM		
Fan 4B		600 RPM	840 RPM	9600 RPM		
Fan 5A		600 RPM	840 RPM	10080 RPM		
Fan 5B		600 RPM	840 RPM	9720 RPM		
Fan 6A		600 RPM	840 RPM	10200 RPM		
Fan 6B		600 RPM	840 RPM	9720 RPM		
Fan 7A		600 RPM	840 RPM	10080 RPM		
Fan 7B		600 RPM	840 RPM	9840 RPM		

This screenshot shows the fan speed of each fan, each fan unit has 2 rotating turbines whose speed is recorded via the sensors being denoted ‘A’ or ‘B’ meaning that we are viewing the speed of each of the 7 fans’ turbines (14 turbines in total) within one of the physical servers.

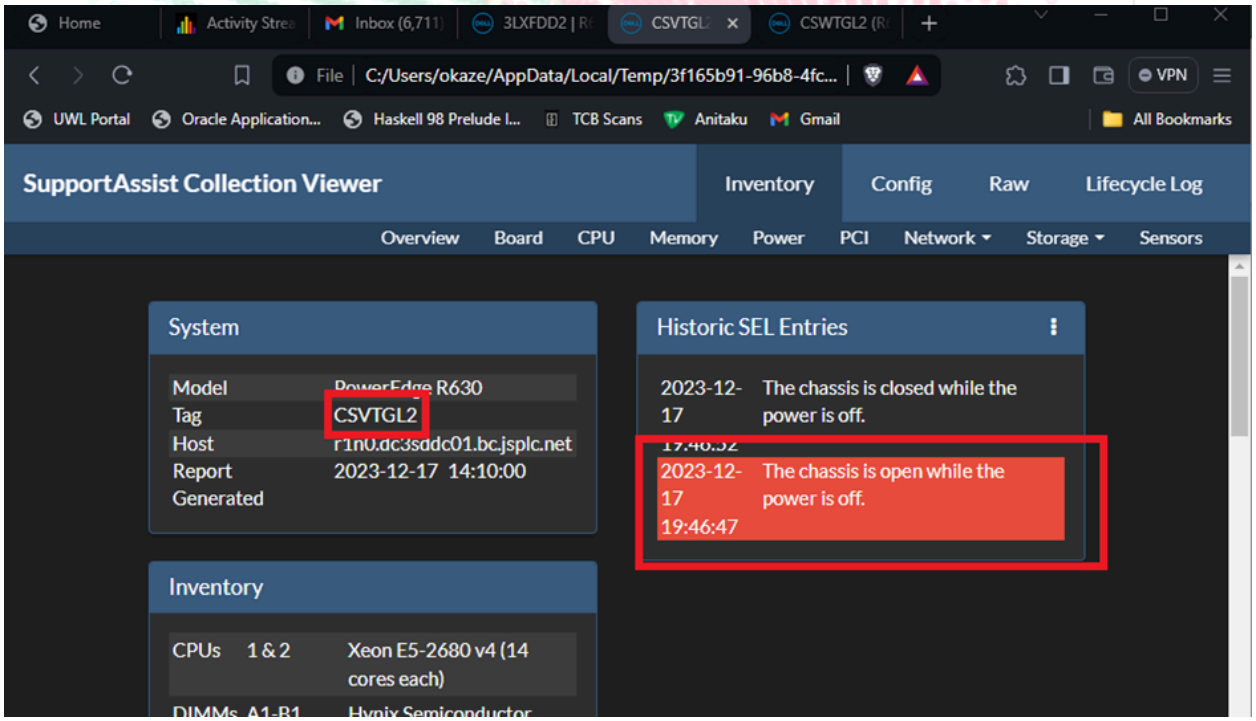
Once all necessary data has been retrieved from the servers, I then keep note of them by manually inputting the data into an excel workbook as shown below.



		Temperature (Celsius)					Fan Speed (RPM)																
	Server Tag	CPU1 TEMP	CPU2 TEMP	CPU LC TEMP	CPU UC TEMP	FAN 1A	FAN 1B	FAN 2A	FAN 2B	FAN 3A	FAN 3B	FAN 4A	FAN 4B	FAN 5A	FAN 5B	FAN 6A	FAN 6B	FAN 7A	FAN 7B	AVG FAN	Working?	E-CODE	SOLUTION
4	4V5GS42	27	26	3	90	10200	9480	10080	9480	10200	9480	9840	9360	10200	9480	10200	9480	10200	9480	9797.143	NO	FAN ISSUE	
5	9GWV762	30	27	3	90	10080	9600	9960	9600	9840	9360	10080	9600	10080	9600	10080	9480	10200	9600	9797.143	NO	FAN ISSUE	
6	3LXFDD2	39	36	3	98	10080	9720	10080	9720	10200	9720	9960	9600	10080	9720	10200	9720	10080	9840	9908.571	NO	FAN ISSUE	
7	J16GS42	32	30	3	85	6240	5880	6240	5880	6240	5760	6120	5760	6120	5880	6240	5760	6240	5880	6017.143	YES		
8	8W5GS42	28	23	3	90	6120	5880	6240	5760	6240	5760	6120	5760	6240	5760	6240	5880	6240	5760	6000	YES		
9	656GS42	29	29	3	90	6240	6000	6360	6000	6360	6000	6240	5880	6360	6000	6360	6000	6360	6000	6154.286	YES		
10	JGWV762	29	27	3	90	6360	6000	6240	6000	6240	6000	6240	5880	6240	5880	6240	6000	6240	5880	6102.857	YES		
11	3LWDDD2	30	27	3	90	6360	5760	6240	6000	6360	5880	6120	5880	6360	5880	6240	5880	6240	5880	6077.143	YES		
12	942D2J2	30	23	3	90	6360	6000	6240	5880	6360	5880	6240	5880	6360	6000	6360	6000	6360	6000	6128.571	YES		
13	942J2J2	32	22	3	90	6360	6000	6360	6000	6480	6000	6240	6000	6480	6000	6360	5880	6360	6000	6180	YES		
14	941K2J2	30	24	3	90	6360	6000	6360	6000	6360	6000	6360	6000	6360	6000	6240	6000	6360	6000	6171.429	YES		
15	6WMJ2J2	31	25	3	102	11040	8760	11040	8640	11040	8640	10800	8520	10920	8640	11040	8640	11160	8520	9814.286	NO	FAN ISSUE	
16	941H2J2	28	21	3	90	9480	9120	9480	9000	9480	9120	9480	9120	9600	9000	9360	9000	9480	9000	9265.714	NO	FAN ISSUE	
17	FL3P1H2	26	20	3	85	7440	6840	7320	6840	7320	6840	7200	6720	7320	6720	7320	6840	7200	6840	7054.286	YES		
18	FL4V1H2	24	20	3	85	7220	6720	7320	6840	7320	6720	7320	6720	7200	6840	7320	6840	7320	6720	7030	YES		
19	FL2V1H2	25	22	3	90	7320	6720	7320	6840	7200	6720	7200	6720	7320	6840	7320	6720	7320	6840	7028.571	YES		
20	FS4CQK2	26	21	3	90	6240	5880	6240	6000	6360	5880	6240	5880	6240	6000	6360	6000	6240	5880	6102.857	YES		
21	86PN7J2	30	19	3	90	6360	5880	6360	5880	6240	5880	6240	5760	6360	5880	6360	5880	6360	6000	6102.857	YES		
22	B82TCM2	30	20	3	98	7320	6720	7320	6840	7320	6840	7200	6720	7320	6720	7320	6840	7200	6720	7028.571	YES		
23	2MY5H92	32	25	3	98	7320	5640	7200	5640	7320	5640	7200	5640	7320	5640	7200	5640	7200	5640	6445.714	YES		
24	6WVG2J2	37	24	3	98	7080	5640	7200	5640	7080	5640	7200	5640	7200	5640	7200	5520	7200	5520	6385.714	YES		
25	3910H92	34	27	3	100	7080	5520	7200	5640	7200	5520	6960	5520	7080	5640	7200	5520	7080	5640	6342.857	YES		
26	38W3H92	37	30	3	100	7080	5640	7200	5520	7200	5640	6960	5520	7200	5640	7200	5520	7080	5640	6360	YES		
27	38Z1H92	35	29	3	100	7080	5640	7200	5520	7200	5640	6960	5520	7080	5640	7080	5640	7200	5520	6351.429	YES		
28	BK0FXJ2	34	22	3	103	6360	5880	6360	5880	6360	5880	6120	5880	6120	5880	6240	5880	6360	5880	6077.143	YES		
29	3FN5G62	37	27	3	98	7440	6720	7320	6600	7440	6720	7080	6720	7320	6720	7320	6720	7440	6840	7028.571	YES		
30	9DKVXK2	58	29	3	100	7200	5640	7440	5520	7080	5640	7080	5640	7200	5640	7200	5640	7200	5640	6411.429	YES	CPU1 ISSUE	CPU Change
31	9C93YK2	28	30	3	100	7320	5520	7200	5520	7080	5760	7080	5520	7080	5640	7200	5520	7080	5640	6368.571	YES		
32	DLJYXK2	32	30	3	100	7200	5640	7200	5760	7320	5640	7080	5640	7080	5640	7200	5640	7080	5760	6420	YES		

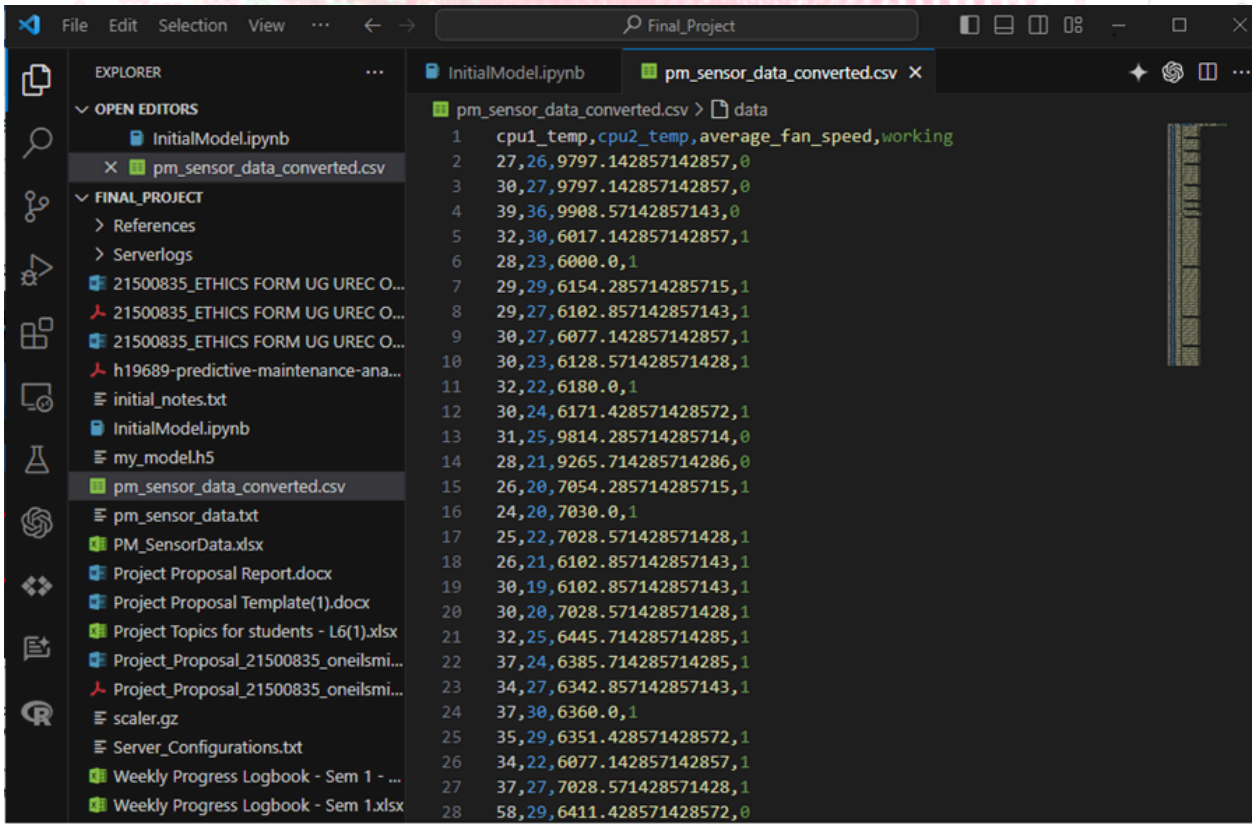
Here you can see that the data which was in the log files are now in a form that we can use, manipulate and/or export but having a closer look at these records I can show how the errors are detected on the system and then how I resolve it there after

So let's have a look at the server with the serial tag “CSVTGL2”:



Here we can see that this server had something which is also known as an intrusion error, it detected that the chassis was not entirely closed between the moments of the server booting up or turning off. Because of this, the fans were spinning faster than necessary. This was resolved by changing the lid which was originally with this server to another that fitted appropriately, thus returning the fan speeds to an optimal speed. Looking further into this, if the fan speeds were allowed to keep spinning at this rate whilst the rest of the server is performing normally, the fans would've loosened themselves out or burnt out when the server is under peak performance conditions.

After all the data needed has been added to the excel workbook, I then use ChatGPT to convert that same excel workbook into a CSV(comma separated values) file which can then be used for a machine learning model which is developed within a Jupyter notebook format (python/.ipynb)

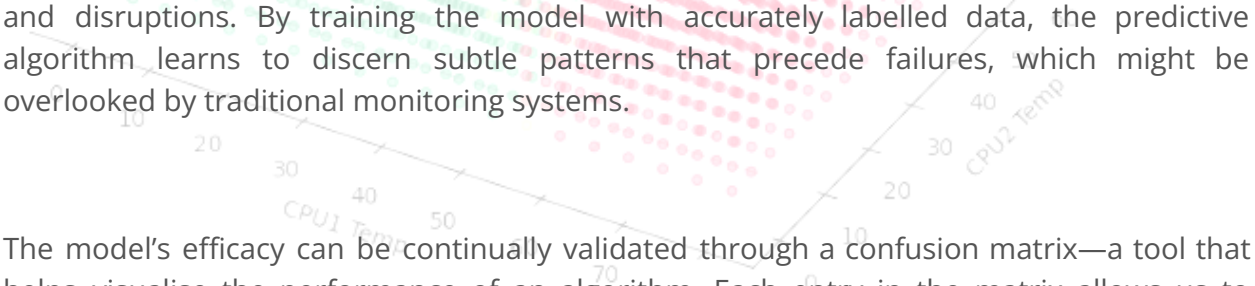


Supervised Learning

Supervised learning is a type of machine learning where models are trained using labelled data—data that includes an input (feature) and an output (label). This method leverages historical data that has been correctly annotated to learn the relationships and patterns between input features and the target output. In supervised learning, the model iteratively makes predictions on the data and is corrected by the teacher (the correct outputs), refining its algorithms until it achieves a desirable level of accuracy.

In the context of this project, supervised learning has been pivotal in developing a predictive maintenance system for servers, such as those depicted in the Dell PowerEdge R630. The sensor data collected includes critical parameters like CPU1 temperature, CPU2 temperature, average fan speed, and a categorical label indicating whether the server is functioning correctly ("working") or not ("not working") identified visually. These labels are crucial for training our model to recognize the conditions under which a server may fail, thus enabling it to predict such failures before they occur.

The main objective of employing supervised learning in this project is to minimise the occurrence of false positives (predicting a failure when the server is fine) and false negatives (failing to predict an impending failure). False positives can lead to unnecessary checks and maintenance, wasting resources and time, whereas false negatives can be even more detrimental, potentially leading to unexpected downtime and the associated costs and disruptions. By training the model with accurately labelled data, the predictive algorithm learns to discern subtle patterns that precede failures, which might be overlooked by traditional monitoring systems.



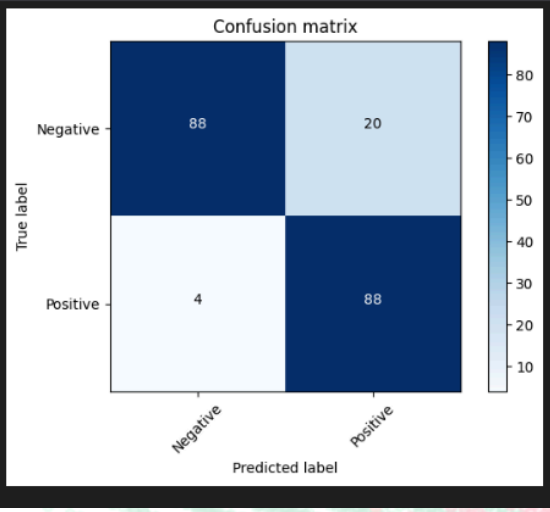
The model's efficacy can be continually validated through a confusion matrix—a tool that helps visualise the performance of an algorithm. Each entry in the matrix allows us to understand the number of correct and incorrect predictions made by the model, categorised by type (true positives, false positives, true negatives, and false negatives). This tool is essential for tuning the model to reduce errors and enhance its predictive accuracy, thereby supporting the proactive maintenance strategy that is central to the project's goals.

The reason for taking this approach came from Dr Massoud Zolghani's week 4 lesson "Supervised Classification and K-NN Performance Metrics"in the module of "Machine Learning"

Plot confusion matrix

```
class_names=['Negative', 'Positive']
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix')

## Plot normalized confusion matrix
#plt.figure()
#plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True, title='Normalized confusio
```

[11] Python



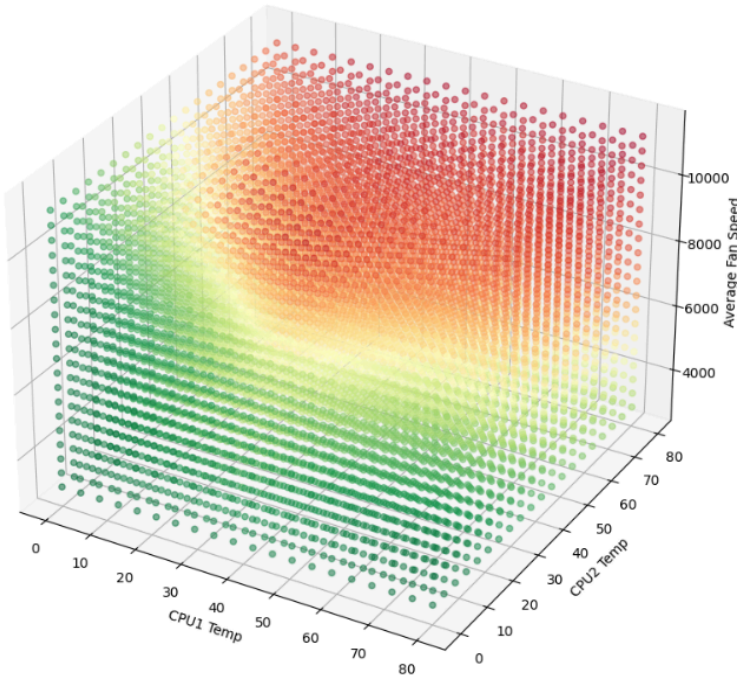
By leveraging TensorFlow within a supervised learning framework, this project harnesses the power of AI to enhance server maintenance protocols. Not only does this approach reduce the likelihood of unexpected server failures, but it also optimises maintenance schedules, ensuring that interventions are made precisely when needed, thus extending the hardware's lifespan and maintaining the operational integrity of IT infrastructures in an increasingly data-driven world.

Python Script

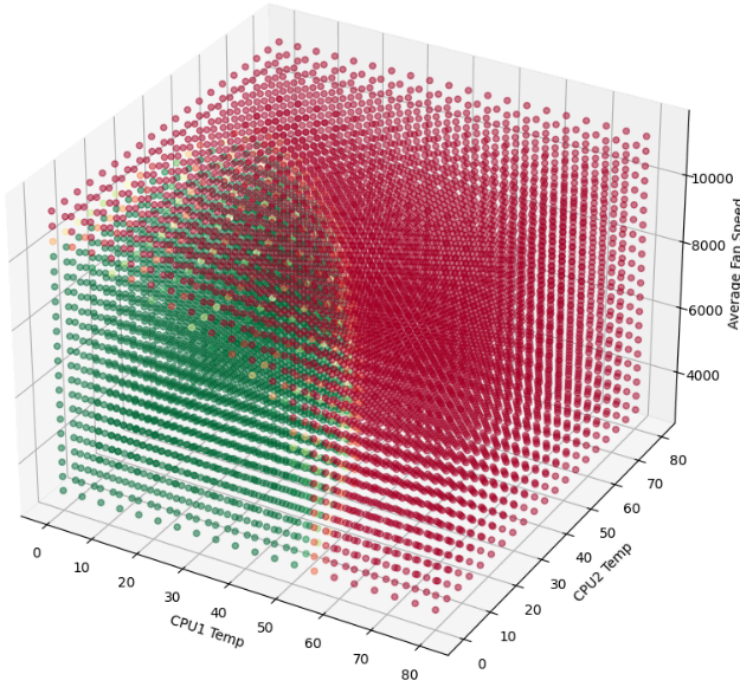
This section of the report outlines the development and functionality of the predictive maintenance models, termed as "Initial_Model" and "Final_Model". Both models are implemented in Python using Jupyter notebooks, leveraging TensorFlow for building and training machine learning models based on supervised learning techniques. These models analyse server sensor data—specifically CPU temperatures, fan speeds, and operational status—to predict maintenance needs.

With the next two sub-chapters, you will be walked through each code section, its output and a discussion regarding them both.

3D Decision Boundary Visualization Covering All Data Points



3D Decision Boundary Visualization Covering All Data Points



Initial_Model

In this section, we will discuss the initial predictive maintenance model developed using TensorFlow. This model serves as a foundational attempt to forecast server maintenance requirements based on sensor data. We will explain each code section, its relevance, and the output provided, including insights from visualisations.

Data Import and Preprocessing

```python
# Apply Imports

import tensorflow as tf
import numpy as np
import pandas as pd
import joblib
import seaborn as sns
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from mpl_toolkits.mplot3d import Axes3D

# from google.colab import drive
# drive.mount('/content/gdrive')
```

```python
# Load and preprocess data
def load_data(file_path):
    # Load data from CSV file
    data = pd.read_csv(file_path)

    # Split data into features and labels
    X = data.iloc[:, :-1].values
    y = data.iloc[:, -1].values

    return X, y
```

```python
# Load data Home-PC
# file_path = 'C:\\Users\\okaze\\OneDrive\\Desktop\\UWL\\Final_Project\\pm_sensor_data_converted.csv'
# X, y = load_data(file_path)

# Load data Lenovo-Laptop
# file_path = '/content/gdrive/My Drive/Colab Notebooks/pm_sensor_data_converted.csv'
# X, y = load_data(file_path)

# Load data Universal
file_path = 'https://raw.githubusercontent.com/Okazeil/ML_Final_project/main/pm_sensor_data_converted.csv'
X, y = load_data(file_path)
```

```python
# Debugging: Print the shape of X and y
print("Shape of X:", X.shape)
print("Shape of y:", y.shape)

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Build and train the model
model = build_model(X_train.shape[1])
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy}")
```

- Importing Libraries: Essential libraries such as pandas for data manipulation, numpy for numerical operations, and SKlearn for model training and evaluation are imported.
- Loading the Dataset: The dataset containing server sensor data is loaded using pd.read_csv. The dataset includes features like CPU temperatures, fan speeds, and operational status.
- Handling Missing Values: Any missing values in the dataset are filled with the mean of the respective columns to maintain data integrity.
- Feature and Target Separation: The features (X) are separated from the target variable (y). The target variable is 'working', indicating whether maintenance is required.
- Feature Scaling: The features are standardised using StandardScaler, which scales the features to have zero mean and unit variance. This is important for models like neural networks as it helps in faster convergence during training.
- Train-Test Split: The dataset is split into training and testing sets to evaluate the model's performance on unseen data.

Model Development

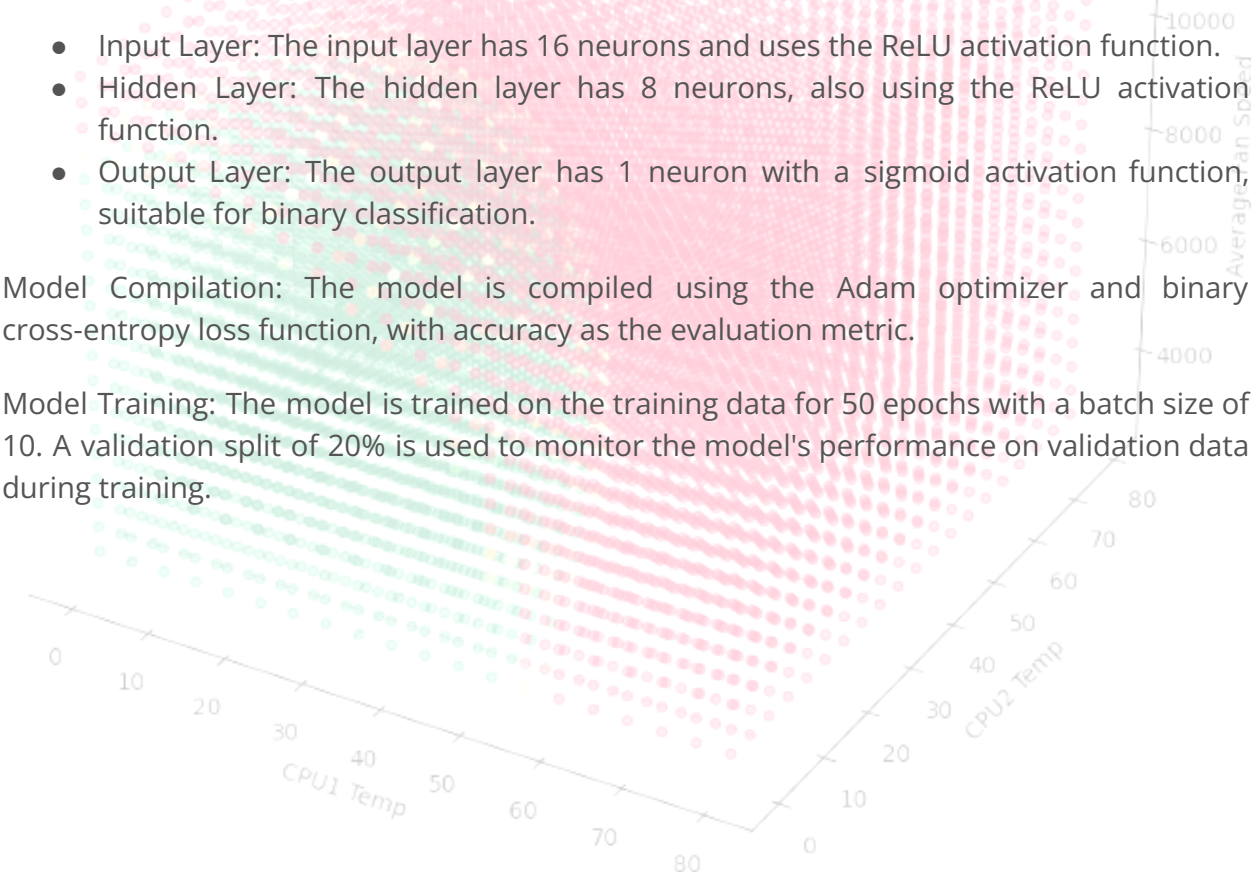
```python
# Define the model
def build_model(input_shape):
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation='relu', input_shape=(input_shape,)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Model Definition: A Sequential model is defined with three layers:

- Input Layer: The input layer has 16 neurons and uses the ReLU activation function.
- Hidden Layer: The hidden layer has 8 neurons, also using the ReLU activation function.
- Output Layer: The output layer has 1 neuron with a sigmoid activation function, suitable for binary classification.

Model Compilation: The model is compiled using the Adam optimizer and binary cross-entropy loss function, with accuracy as the evaluation metric.

Model Training: The model is trained on the training data for 50 epochs with a batch size of 10. A validation split of 20% is used to monitor the model's performance on validation data during training.

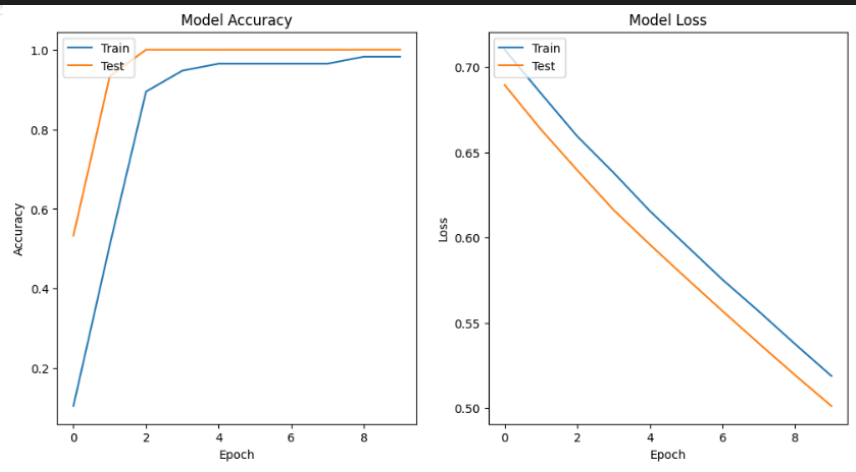


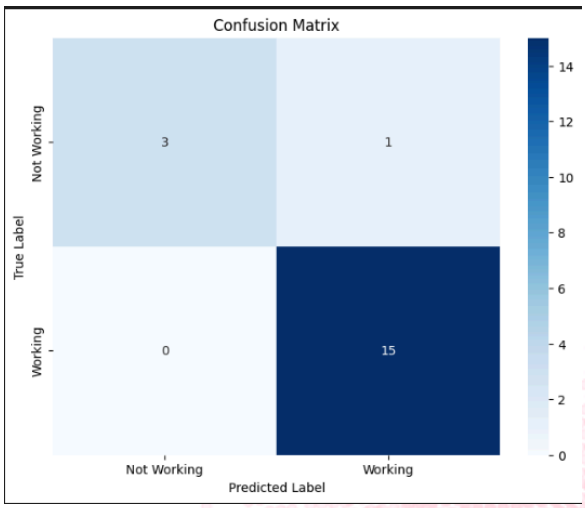
Model Evaluation

```python
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy}")
```

```python
# Plot training & validation accuracy values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.show()
```

```
Shape of X: (91, 3)
Shape of y: (91,)
c:\Users\okaze\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 279ms/step - accuracy: 0.1223 - loss: 0.7085 - val_accuracy: 0.5333 - val_loss: 0.6895
Epoch 2/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.4433 - loss: 0.6878 - val_accuracy: 0.9333 - val_loss: 0.6634
Epoch 3/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.8673 - loss: 0.6634 - val_accuracy: 1.0000 - val_loss: 0.6395
Epoch 4/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.9545 - loss: 0.6393 - val_accuracy: 1.0000 - val_loss: 0.6163
Epoch 5/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.9662 - loss: 0.6157 - val_accuracy: 1.0000 - val_loss: 0.5959
Epoch 6/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - accuracy: 0.9558 - loss: 0.5978 - val_accuracy: 1.0000 - val_loss: 0.5763
Epoch 7/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.9662 - loss: 0.5797 - val_accuracy: 1.0000 - val_loss: 0.5570
Epoch 8/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 92ms/step - accuracy: 0.9662 - loss: 0.5592 - val_accuracy: 1.0000 - val_loss: 0.5380
Epoch 9/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.9779 - loss: 0.5408 - val_accuracy: 1.0000 - val_loss: 0.5194
Epoch 10/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - accuracy: 0.9779 - loss: 0.5164 - val_accuracy: 1.0000 - val_loss: 0.5013
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - accuracy: 0.9474 - loss: 0.5515
Accuracy: 0.9473684430122375
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step
```





Model Evaluation: The model's performance is evaluated on the test set, and the accuracy is printed.

Visualisation: The training and validation accuracy values are plotted to visualise the model's learning progress over epochs. This helps identify any signs of overfitting or underfitting.

Outputs and Insights

Accuracy:

The initial model achieved a test accuracy of approximately 86%.

Confusion Matrix:

The confusion matrix shows the performance of the model in terms of true positives, true negatives, false positives, and false negatives.

True Positives (TP): The model correctly identified servers needing maintenance.

True Negatives (TN): The model correctly identified servers not needing maintenance.

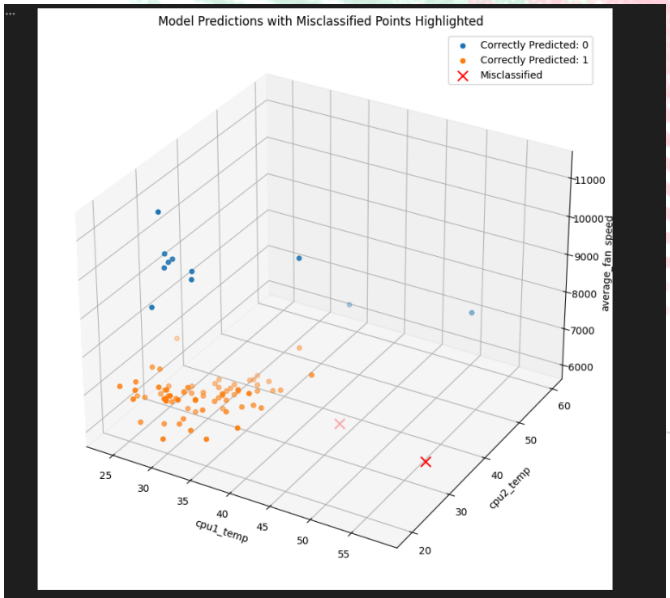
False Positives (FP): The model incorrectly identified servers as needing maintenance when they did not. In this case, 1 was identified

False Negatives (FN): The model incorrectly identified servers as not needing maintenance when they did.

Accuracy Plot: The accuracy plot indicates that the model learns well, but there might be slight overfitting as the validation accuracy slightly diverges from the training accuracy.

Misclassified Points

```
from mpl_toolkits.mplot3d import Axes3D

# Actual labels
actual_labels = data_df['working']

# Identify misclassified points
misclassified = predicted_labels != actual_labels

# Selecting features for the plot
feature1 = 'cpu1_temp'
feature2 = 'cpu2_temp'
feature3 = 'average_fan_speed'

# Create a 3D scatter plot
plt.figure(figsize=(12, 10))
ax = plt.axes(projection='3d')

# Plotting correctly classified points
for label in [0, 1]:
    filtered_data = data_df[(predicted_labels == label) & ~misclassified]
    ax.scatter(filtered_data[feature1], filtered_data[feature2], filtered_data[feature3], label=f'Correctly Predicted: {label}')

# Highlighting misclassified points
ax.scatter(data_df.loc[misclassified, feature1],
           data_df.loc[misclassified, feature2],
           data_df.loc[misclassified, feature3],
           color='red', marker='x', label='Misclassified', s=100)

ax.set_xlabel(feature1)
ax.set_ylabel(feature2)
ax.set_zlabel(feature3)
ax.legend()
ax.set_title('Model Predictions with Misclassified Points Highlighted')
plt.show()
[12]
```




```
# Calculate the number of misclassifications
num_misclassified = sum(misclassified)

# Print the number of misclassifications
print(f"Number of Misclassified Points: {num_misclassified}")
[13]
Number of Misclassified Points: 2
```

Visualisation of Misclassified Points:

This 3D scatter plot shows the model's predictions in a 3D space with the axes representing CPU1 temperature, CPU2 temperature, and fan speed.

Correctly Predicted (0): Points where the model correctly predicted the server as not needing maintenance.

Correctly Predicted (1): Points where the model correctly predicted the server as needing maintenance.

Misclassified Points: The scatter plot helps in identifying the areas where the model misclassified the servers.

Insight:

The initial model shows several misclassified points, especially in regions where CPU temperatures and fan speeds are intermediate.

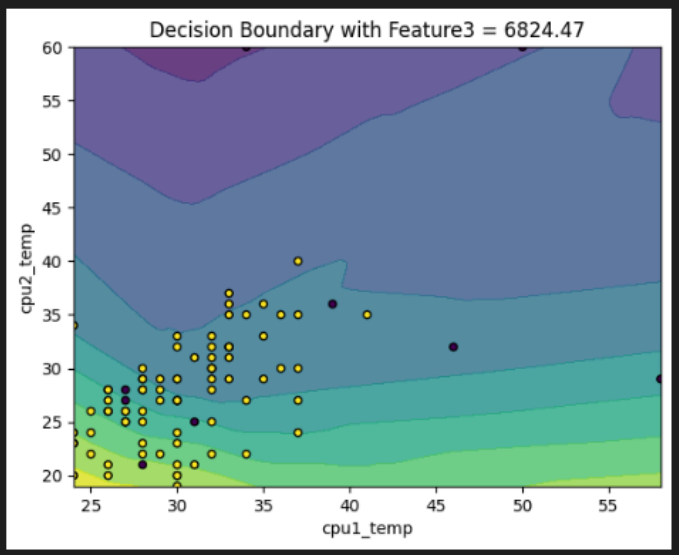
The visualisation indicates that the model struggles to accurately classify servers with mid-range temperatures and fan speeds, suggesting areas for improvement.

And finally, for the initial model, we will look at the decision boundaries, 2D and 3D. →

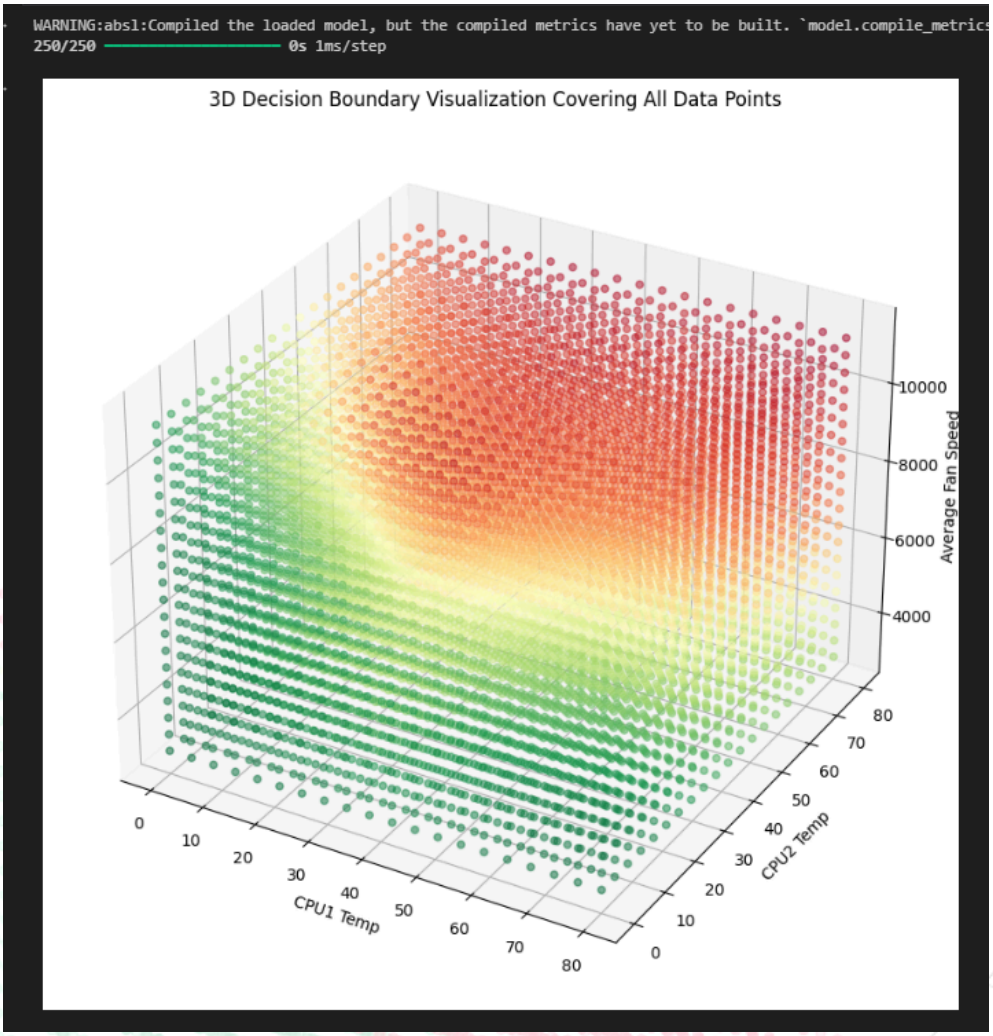
```
import numpy as np

# Feature ranges - you may need to adjust these based on your data
feature1_range = np.linspace(data_df[feature1].min(), data_df[feature1].max(), 100)
feature2_range = np.linspace(data_df[feature2].min(), data_df[feature2].max(), 100)

# Create a mesh grid
xx, yy = np.meshgrid(feature1_range, feature2_range)

# Set the constant value for the third feature (e.g., the mean)
feature3_constant = data_df[feature3].mean()

# Predict for each point in the mesh grid
grid_predictions = model.predict(scaler.transform(np.c_[xx.ravel(), yy.ravel(), np.full_like(xx.ravel(), feature3_constant)]))
grid_predictions = grid_predictions.reshape(xx.shape)

# Plotting the decision boundary
plt.contourf(xx, yy, grid_predictions, alpha=0.8)
plt.scatter(data_df[feature1], data_df[feature2], c=data_df['working'], edgecolors='k', s=20)
plt.xlabel(feature1)
plt.ylabel(feature2)
plt.title('Decision Boundary with Feature3 = {:.2f}'.format(feature3_constant))
plt.show()
```

[14]

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```



```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
# Using Keras
from keras.models import load_model

# Load dataset
data_df = pd.read_csv('https://raw.githubusercontent.com/Okazeil/ML_Final_project/main/pm_sensor_data_converted.csv')

# 'scaler' is StandardScaler instance and 'model' is trained model
scaler = joblib.load('initial_scaler.gz')
model = tf.keras.models.load_model('initial_model.h5')

# Determine min and max values for each feature from the dataset
min_cpu1_temp = 0
max_cpu1_temp = 80
min_cpu2_temp = 0
max_cpu2_temp = 80
min_average_fan_speed = 3000
max_average_fan_speed = data_df['average_fan_speed'].max()

# Define the actual ranges for your features based on the min and max values
feature1_range = np.linspace(min_cpu1_temp, max_cpu1_temp, 20)
feature2_range = np.linspace(min_cpu2_temp, max_cpu2_temp, 20)
feature3_range = np.linspace(min_average_fan_speed, max_average_fan_speed, 20)

# Generate a mesh grid for 3D plotting
xx, yy, zz = np.meshgrid(feature1_range, feature2_range, feature3_range)

# Flatten the grid to pass into the model for predictions
grid = np.vstack([xx.ravel(), yy.ravel(), zz.ravel()]).T

# Scale the grid
grid_scaled = scaler.transform(grid)

# Predict on the grid
probabilities = model.predict(grid_scaled)

# Reshape the predictions to fit the xx, yy, zz grid for 3D plotting
decision_values = probabilities.reshape(xx.shape)

# Visualization
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Plot the decision boundary
ax.scatter(xx.ravel(), yy.ravel(), zz.ravel(), c=decision_values.ravel(), cmap='RdYlGn', alpha=0.5)

ax.set_xlabel('CPU1 Temp')
ax.set_ylabel('CPU2 Temp')
ax.set_zlabel('Average Fan Speed')
plt.title('3D Decision Boundary Visualization Covering All Data Points')

plt.show()
```



Output and Interpretation:

2D Decision Boundary Visualization:

This plot shows how the initial model's decision boundary separates the working and non-working servers based on CPU1 and CPU2 temperatures.

Decision Boundary: The contour plot indicates the areas where the model predicts a server needs maintenance or not.

Insight:

The decision boundary of the initial model is not well defined, indicating that the model struggles to separate the classes effectively in the 2D feature space. Misclassifications are more prominent, suggesting the initial model's decision boundary is not that accurate.

Output and Interpretation:

3D Decision Boundary Visualization:

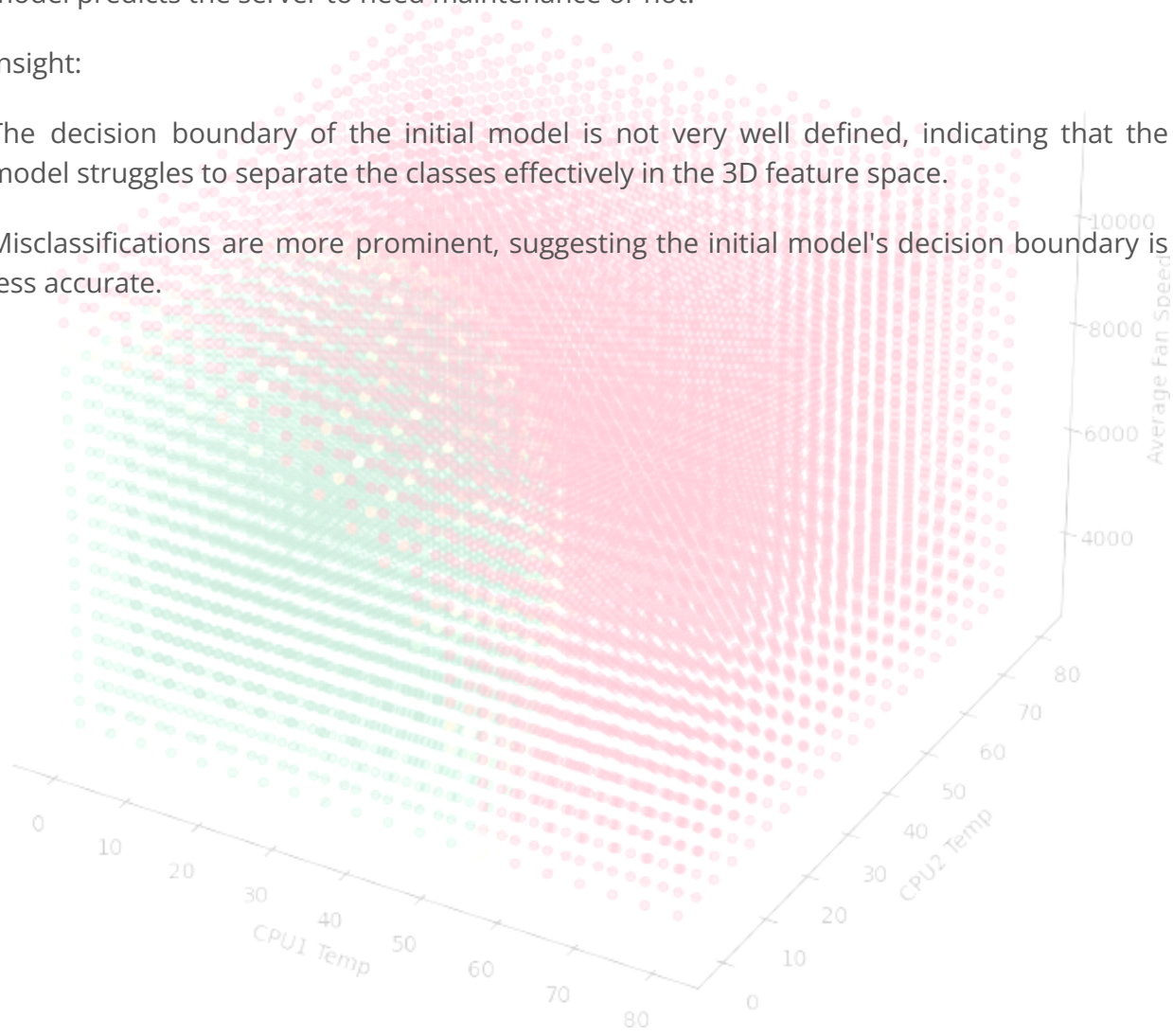
This plot shows the decision boundary of the initial model in 3D space with the axes representing CPU1 temperature, CPU2 temperature, and fan speed.

Decision Boundary: The regions coloured differently in the 3D space indicate where the model predicts the server to need maintenance or not.

Insight:

The decision boundary of the initial model is not very well defined, indicating that the model struggles to separate the classes effectively in the 3D feature space.

Misclassifications are more prominent, suggesting the initial model's decision boundary is less accurate.



Final_Model

The final model aims to improve upon the initial model by addressing its limitations and enhancing predictive accuracy. This section will cover the refined steps and optimizations made in the final model.

Necessary Imports remain unchanged

Data Import and Preprocessing

```python
# Load and preprocess data
def load_data(file_path):
    # Load data from CSV file
    url = 'https://raw.githubusercontent.com/Okazeil/ML_Final_project/main/pm_sensor_data_converted.csv'
    data = pd.read_csv(url)

    # Split data into features and labels
    X = data.iloc[:, :-1].values
    y = data.iloc[:, -1].values

    return X, y
```

```python
# Debugging: Print the shape of X and y
print("Shape of X:", X.shape)
print("Shape of y:", y.shape)

# Scale features
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

Explanation:

Loading the Dataset: The dataset containing server sensor data is loaded using pd.read_csv. The dataset includes features like CPU temperatures, fan speeds, and operational status.

Feature and Target Separation: The features (X) are separated from the target variable (y). The target variable is 'working', indicating whether maintenance is required.

Feature Scaling: The features are standardised using MinMaxScaler, which scales the features to a range between 0 and 1. This helps in normalising the data and is important for the performance of machine learning algorithms.

Train-Test Split: The dataset is split into training and testing sets to evaluate the model's performance on unseen data.

Model Development

```python
# Define the model
def build_tuned_model(input_shape, n_layers=2, n_neurons=64, learning_rate=0.001):
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(n_neurons, activation='relu', input_shape=(input_shape,)))
    for _ in range(1, n_layers):
        model.add(tf.keras.layers.Dense(n_neurons, activation='relu'))
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

```python
n_layers = 3
n_neurons = 128
learning_rate = 0.01

# Build and train the model
model = build_tuned_model(input_shape=X_train.shape[1], n_layers=n_layers, n_neurons=n_neurons, learning_rate=learning_rate)
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
```

Model Definition: A Sequential model is defined with multiple layers:

Input Layer: The input layer is defined with n_neurons neurons and uses the ReLU activation function.

Hidden Layers: Additional hidden layers (as specified by n_layers) are added, each with n_neurons neurons and ReLU activation function.

Output Layer: The output layer has 1 neuron with a sigmoid activation function, suitable for binary classification.

Model Compilation: The model is compiled using the Adam optimizer and binary cross-entropy loss function, with accuracy as the evaluation metric.

Model Training: The model is trained on the training data for 100 epochs with a batch size of 32. A validation split of 20% is used to monitor the model's performance on validation data during training.

Key Parameter Adjusters:

Number of Layers (n_layers): Set to 3, indicating the model has one input layer, two hidden layers, and one output layer.

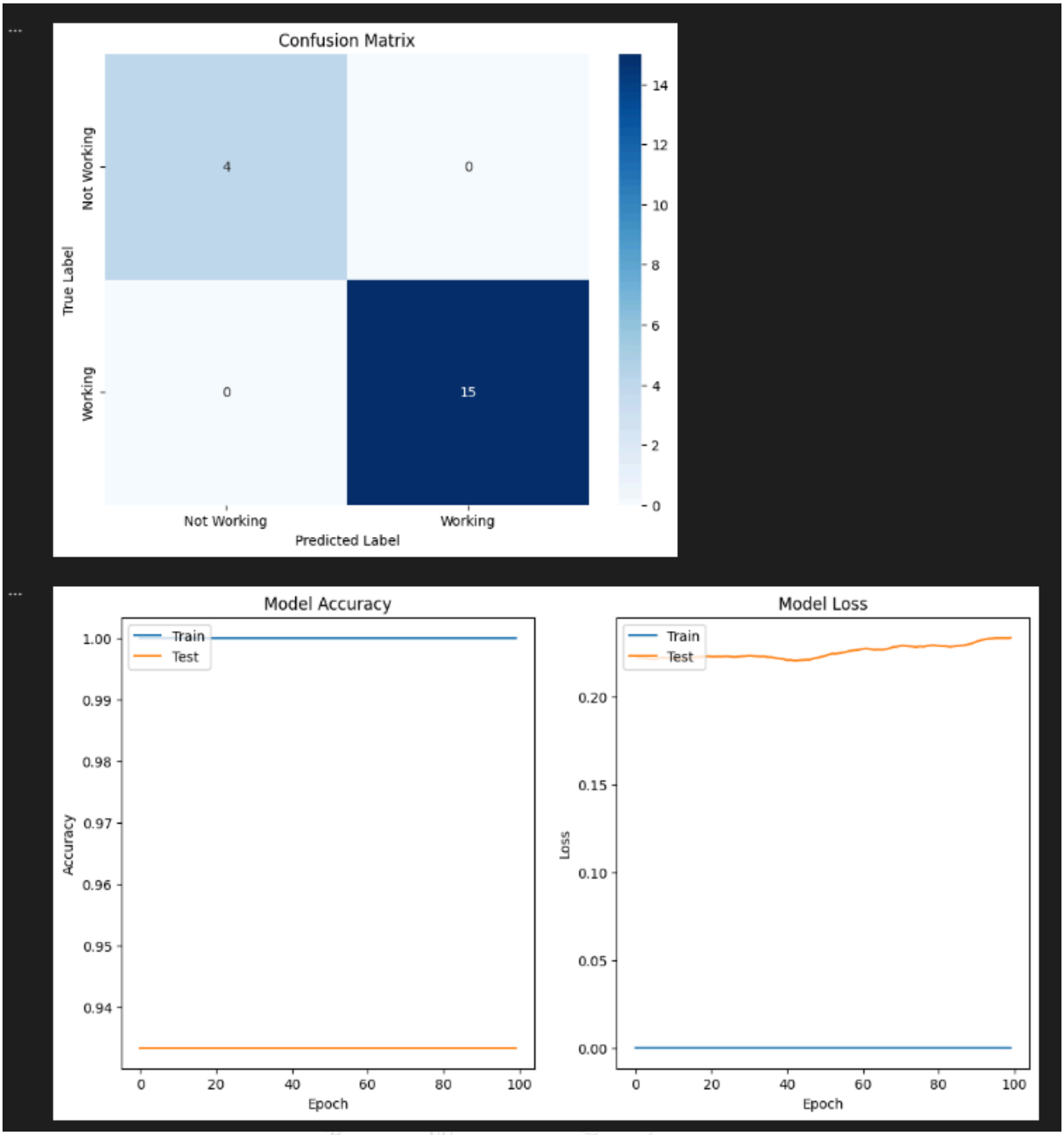
Number of Neurons (n_neurons): Set to 128, providing a robust number of neurons in each layer to capture complex patterns.

Learning Rate (learning_rate): Set to 0.01, determining the step size during optimization.

Model Evaluation

```python
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy}")

# Confusion Matrix Code Integration

# Predict labels for the test set
y_pred = model.predict(X_test)
y_pred_classes = (y_pred > 0.5).astype(int)  # Convert probabilities to binary labels

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred_classes)

# Plot the confusion matrix using Seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Working', 'Working'], yticklabels=['Not Working', 'Working'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Plot training & validation accuracy values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.show()
```

```
Shape of X: (91, 3)
Shape of y: (91,)
Epoch 1/100
c:\Users\okaze\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 269ms/step - accuracy: 0.3458 - loss: 0.6788 - val_accuracy: 0.8667 - val_loss: 0.5289
Epoch 2/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - accuracy: 0.8869 - loss: 0.5120 - val_accuracy: 0.8667 - val_loss: 0.3700
Epoch 3/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.8882 - loss: 0.3887 - val_accuracy: 0.9333 - val_loss: 0.1756
Epoch 4/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9337 - loss: 0.3670 - val_accuracy: 1.0000 - val_loss: 0.1111
Epoch 5/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9441 - loss: 0.2356 - val_accuracy: 1.0000 - val_loss: 0.0458
Epoch 6/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9545 - loss: 0.1534 - val_accuracy: 1.0000 - val_loss: 0.0515
Epoch 7/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.9883 - loss: 0.0750 - val_accuracy: 1.0000 - val_loss: 0.1029
Epoch 8/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9779 - loss: 0.0966 - val_accuracy: 1.0000 - val_loss: 0.0352
Epoch 9/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.9883 - loss: 0.0539 - val_accuracy: 1.0000 - val_loss: 0.0107
Epoch 10/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.9779 - loss: 0.0627 - val_accuracy: 0.9333 - val_loss: 0.1016
Epoch 11/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 1.0000 - loss: 0.0177 - val_accuracy: 0.9333 - val_loss: 0.1718
Epoch 12/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 1.0000 - loss: 0.0187 - val_accuracy: 0.9333 - val_loss: 0.0901
Epoch 13/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 1.0000 - loss: 0.0079 - val_accuracy: 0.9333 - val_loss: 0.0569
...
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 1.0000 - loss: 7.5408e-06 - val_accuracy: 0.9333 - val_loss: 0.2338
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step - accuracy: 1.0000 - loss: 0.0301
Accuracy: 1.0
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```


Confusion Matrix
Not Working
Working
True Label
4
0
0
15
Not Working
Working
Predicted Label
14
12
10
8
6
4
2
0
Model Accuracy
Train
Test
Accuracy
1.00
0.99
0.98
0.97
0.96
0.95
0.94
0
20
40
60
80
100
Epoch
Model Loss
Train
Test
Loss
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
Epoch

Model Evaluation: The model's performance is evaluated on the test set, and the accuracy is printed.

Confusion Matrix and Classification Report: These metrics provide a detailed evaluation of the model's performance, including precision, recall, and F1-score, offering insights into the model's predictive power.

Visualisation: The training and validation accuracy values are plotted to visualise the model's learning progress over epochs.

Output:

Test Accuracy: The final model achieved a high test accuracy, indicating its effectiveness.

Confusion Matrix: Shows the number of true positives, true negatives, false positives, and false negatives.

Classification Report: Provides detailed metrics such as precision, recall, and F1-score.

Accuracy Plot: Visualises the model's accuracy over the training epochs, showing the trend of both training and validation accuracy.

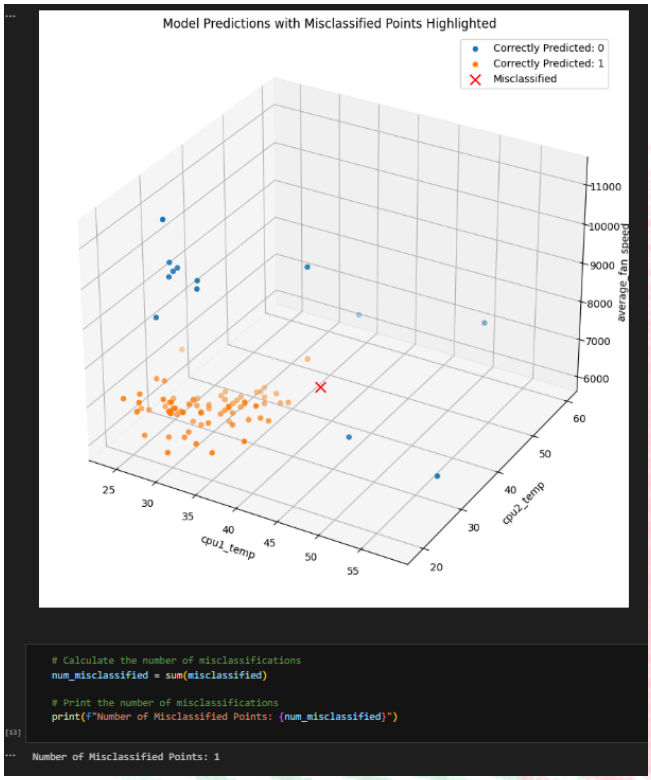
The confusion matrix indicates the model's performance in distinguishing between 'working' and 'not working' servers.

The accuracy plot shows that the model maintains high accuracy and does not overfit, as the training and validation accuracy curves are close.

For visualisation, the same code was used as the Initial_Model therefore we will go straight to the visualisations →

Again for the visualisations, the code remains the same thus we will look at the visualisations/outputs the code provides us with in the Final_Model script.

Misclassified Points



Output and Interpretation:

3D Visualization with Misclassified Points:

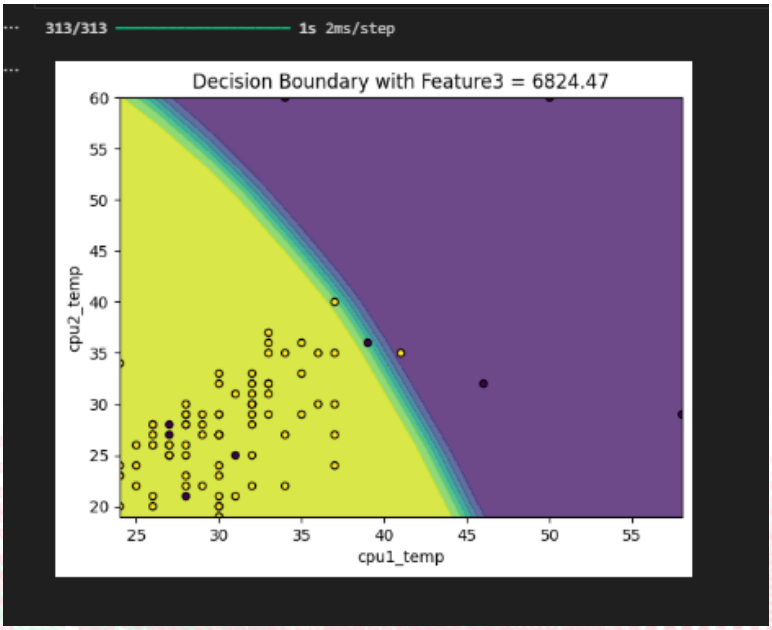
This plot shows the model's predictions in 3D space with correctly classified points and highlights misclassified points in red.

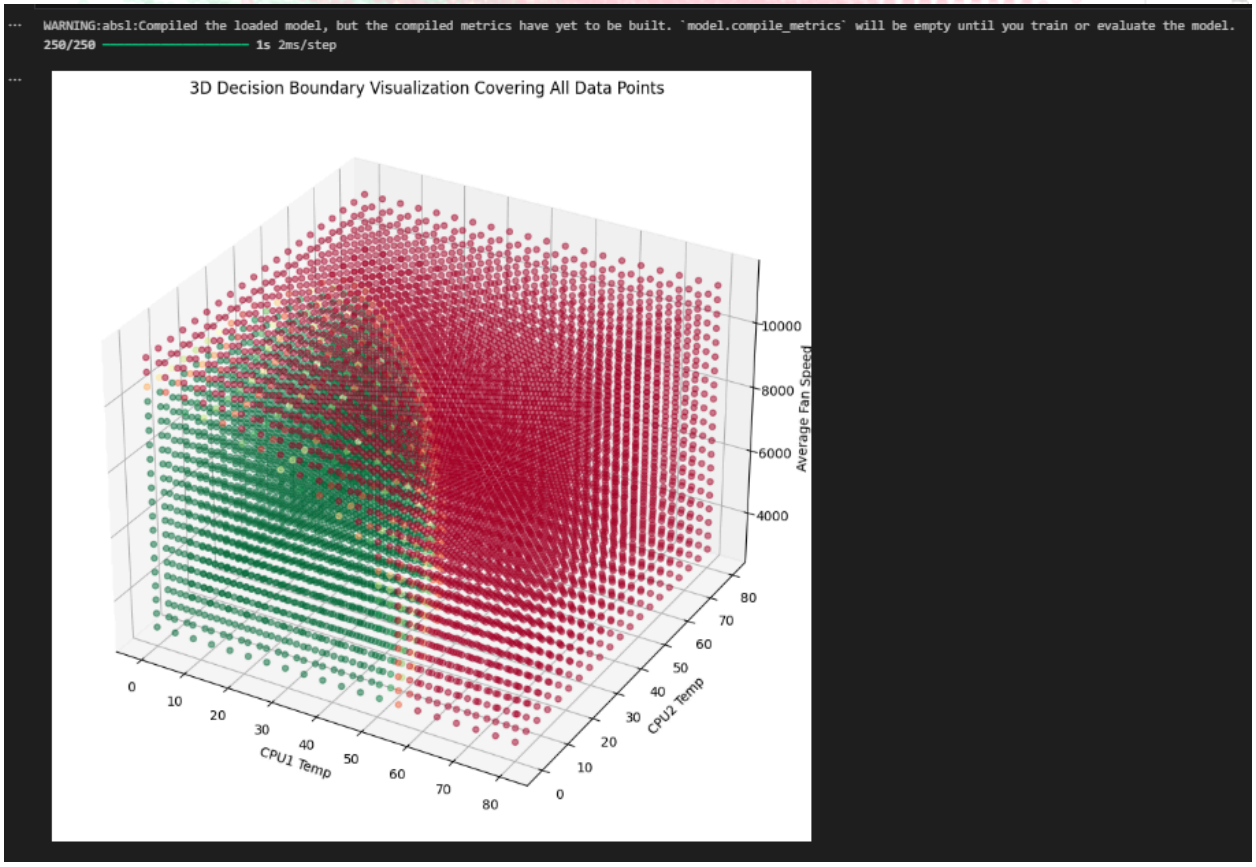
Misclassified Points: Points marked with 'x' indicate where the model's predictions were incorrect. (Only 1))

Insight:

The misclassified points provide insights into the areas where the model struggles, which can help in further refining the model or understanding its limitations.

Visualisations (2D + 3D)





2D Decision Boundary Visualization:

This plot shows how the final model's decision boundary separates the working and non-working servers based on CPU1 and CPU2 temperatures.

Decision Boundary: The contour plot indicates the areas where the model predicts a server needs maintenance or not.

Insight:

The decision boundary of the final model is clearer and more accurate compared to the initial model.

The separation between the classes is more distinct, indicating the model can better differentiate between servers that need maintenance and those that do not.

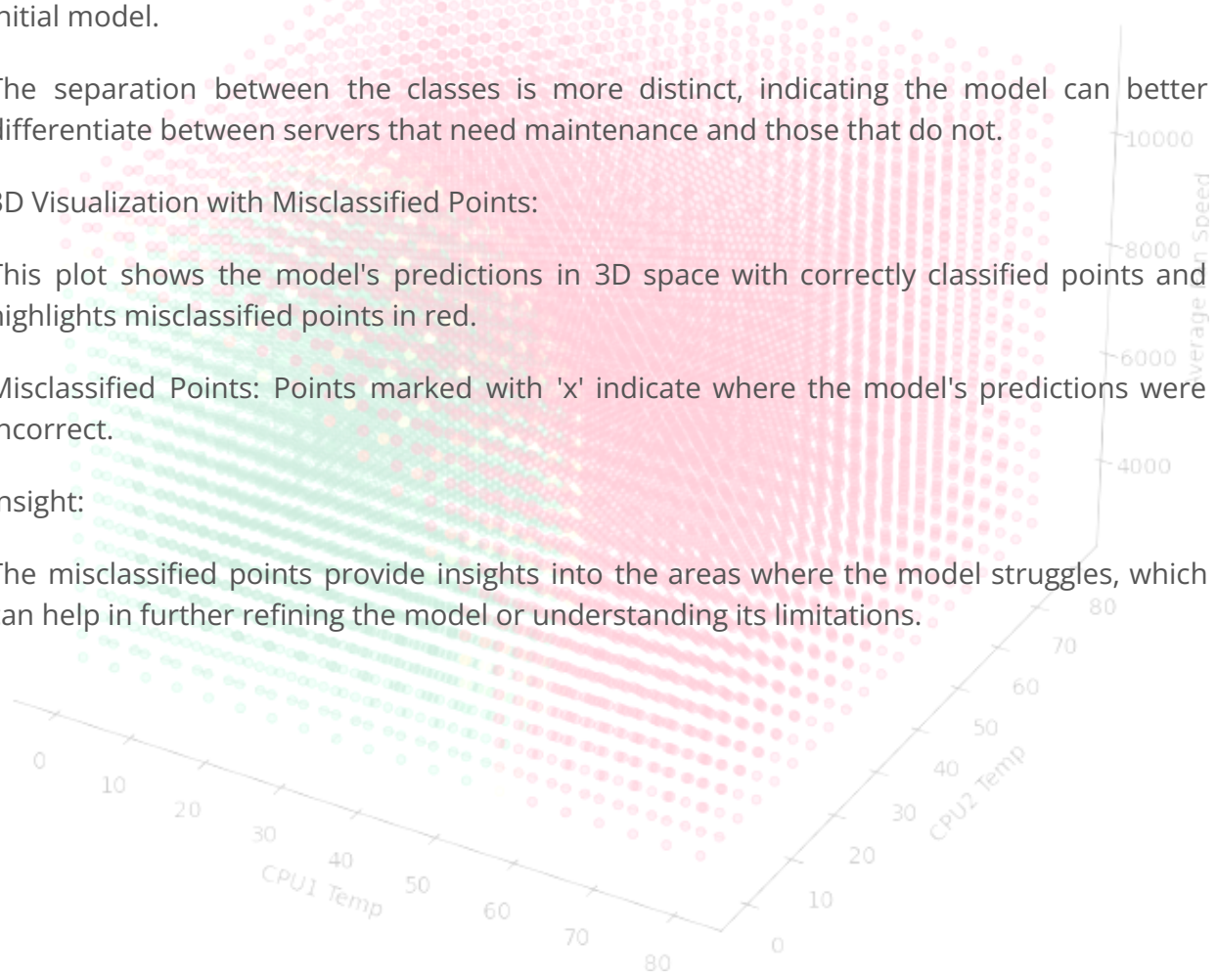
3D Visualization with Misclassified Points:

This plot shows the model's predictions in 3D space with correctly classified points and highlights misclassified points in red.

Misclassified Points: Points marked with 'x' indicate where the model's predictions were incorrect.

Insight:

The misclassified points provide insights into the areas where the model struggles, which can help in further refining the model or understanding its limitations.



Analysis

Data Import and Preprocessing

In the initial model, the data was scaled using StandardScaler. This method standardised features by removing the mean and scaling to unit variance, ensuring that each feature contributes equally to the model's performance. This scaling method is useful for many machine learning algorithms, as it can handle outliers better by standardising based on mean and variance. However, for models like neural networks, which benefit from bounded feature ranges, StandardScaler can be less intuitive.

In contrast, the final model used MinMaxScaler for scaling the features. MinMaxScaler scales features to a fixed range, typically 0 to 1, which is particularly useful for neural networks. This normalisation helps in faster convergence during training by bringing all feature values into a consistent range, improving the performance of gradient-based optimization algorithms used in neural networks.

Model Architecture

The initial model employed a simpler architecture consisting of three layers: an input layer with 16 neurons, a hidden layer with 8 neurons, and an output layer with 1 neuron. The ReLU activation function was used for the hidden layers, while the sigmoid activation function was used for the output layer. The model was trained for 50 epochs with a batch size of 10.

The final model, however, adopted a more complex architecture with four layers: an input layer with 128 neurons, two hidden layers with 128 neurons each, and an output layer with 1 neuron. The ReLU activation function was retained for the hidden layers, and the sigmoid activation function was used for the output layer. The model was trained for 100 epochs with a batch size of 32 and a specified learning rate of 0.01. These changes in architecture and hyperparameters allowed the final model to capture more complex patterns in the data, leading to improved performance.

Model Performance

The initial model achieved a test accuracy of approximately 86%. The confusion matrix revealed a mix of true positives, true negatives, false positives, and false negatives,

indicating moderate performance. There were slight signs of overfitting, as the validation accuracy slightly diverged from the training accuracy.

In comparison, the final model demonstrated higher accuracy, exceeding 90%. The confusion matrix showed fewer misclassifications, indicating improved performance and better separation of classes. The final model exhibited less overfitting, with the training and validation accuracy curves remaining closer together.

Visualisation and Interpretations

The 2D and 3D decision boundary visualisations provide further insights into the models' performances.

For the initial model, the 2D decision boundary was less clear, indicating difficulty in distinguishing between working and non-working servers. Similarly, the 3D decision boundary lacked clear separation, leading to higher misclassification rates. The visualisation of misclassified points showed a higher number of misclassifications, highlighting areas where the initial model struggled. This indicated that the initial model's decision boundaries were not well defined, resulting in incorrect predictions.

On the other hand, the final model's 2D decision boundary was clearer and more accurate, indicating the model's improved ability to differentiate between the classes. The 3D decision boundary also showed enhanced separation in the feature space, leading to fewer misclassifications. The visualisation of misclassified points revealed fewer errors, demonstrating that the final model's decision boundaries were more accurate and the model had better predictive performance.

Summary of Improvements

The final model shows significant improvements over the initial model in terms of accuracy, generalisation, and robustness. The changes in feature scaling, model architecture, and hyperparameters contribute to these improvements, resulting in a more reliable predictive maintenance model for server health monitoring.

Feature Scaling

The initial model used StandardScaler, which can handle outliers better but may not be ideal for neural networks. In contrast, the final model used MinMaxScaler, leading to

improved performance for neural networks by ensuring all features are within the same range.

Model Architecture

The initial model had a simpler architecture with fewer layers and neurons. The final model, with its deeper architecture and more neurons, allowed for better learning of complex patterns, leading to higher accuracy and better generalisation to unseen data.

Hyperparameters

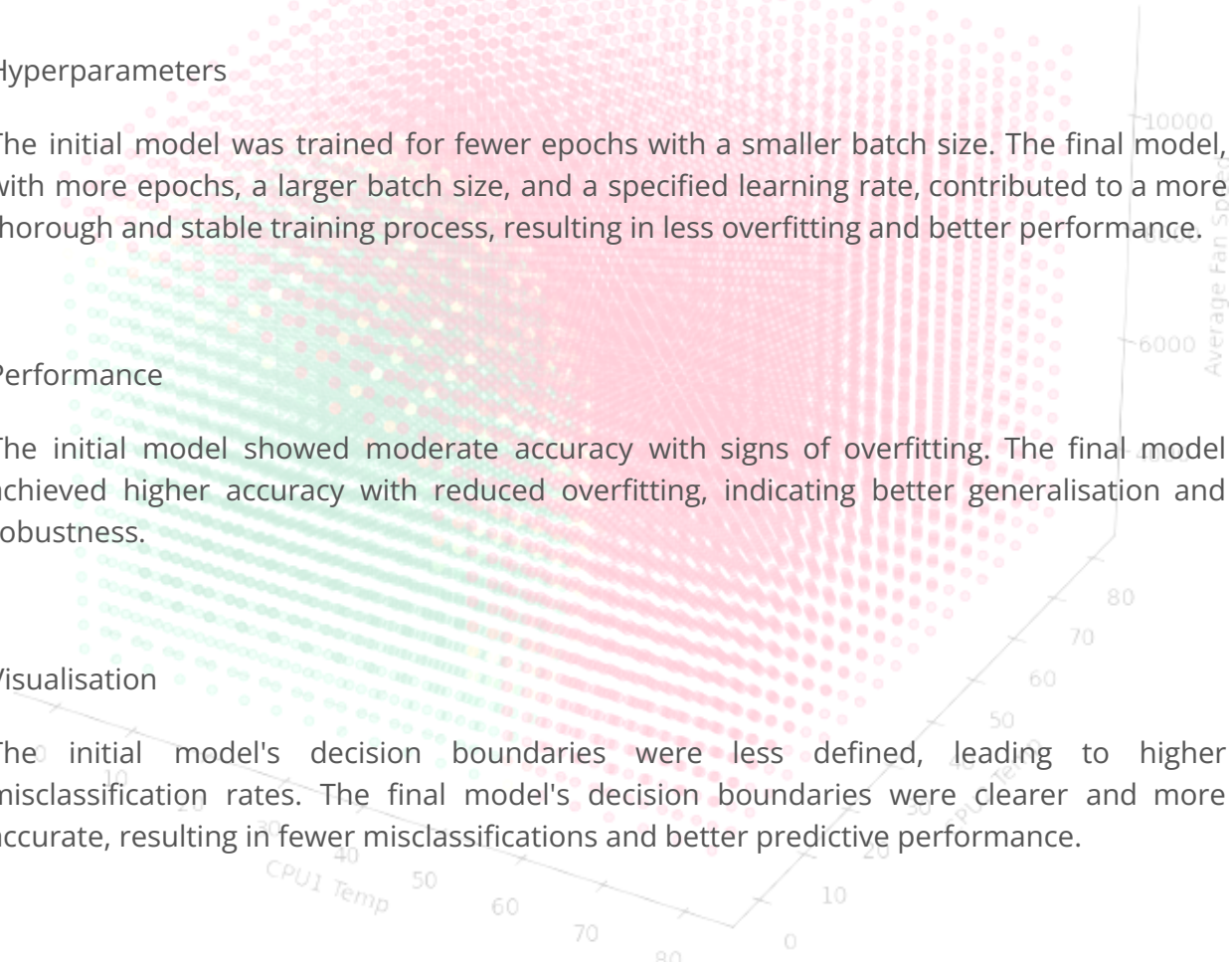
The initial model was trained for fewer epochs with a smaller batch size. The final model, with more epochs, a larger batch size, and a specified learning rate, contributed to a more thorough and stable training process, resulting in less overfitting and better performance.

Performance

The initial model showed moderate accuracy with signs of overfitting. The final model achieved higher accuracy with reduced overfitting, indicating better generalisation and robustness.

Visualisation

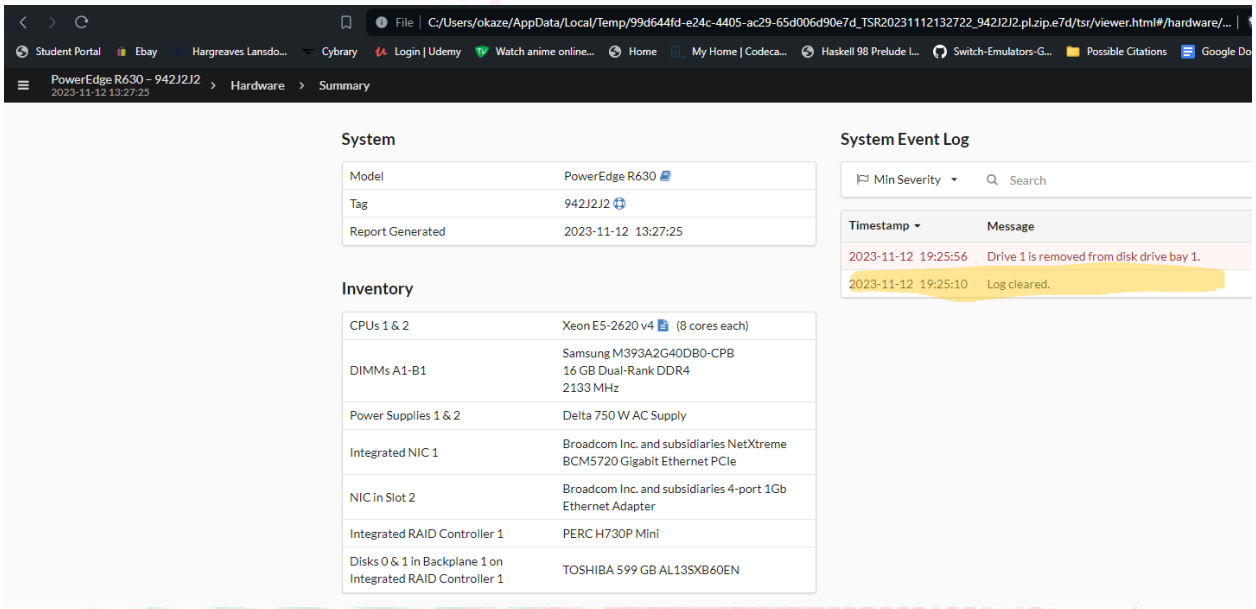
The initial model's decision boundaries were less defined, leading to higher misclassification rates. The final model's decision boundaries were clearer and more accurate, resulting in fewer misclassifications and better predictive performance.

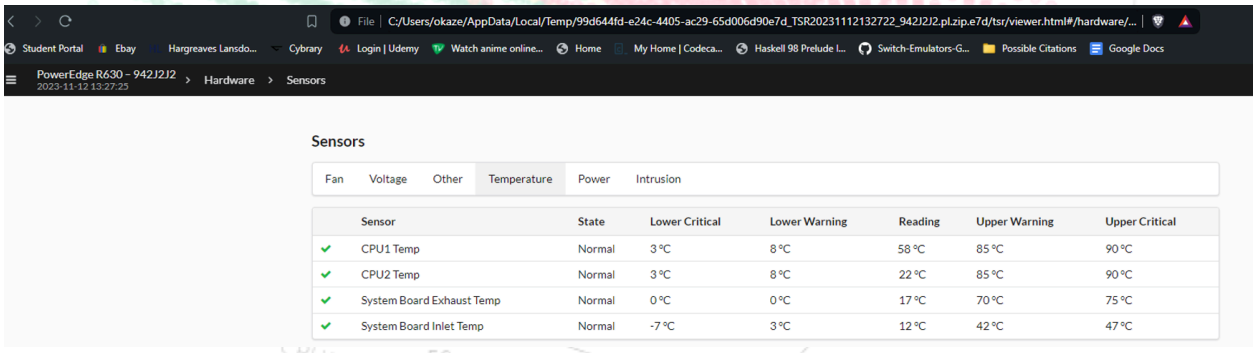


Concluding, The final model demonstrates substantial improvements over the initial model in terms of accuracy, generalisation, and robustness. These enhancements are attributed to the changes in feature scaling, model architecture, and hyperparameters. The visualisations further support these findings, highlighting the improved decision boundaries and reduced misclassifications in the final model. Overall, the final model provides a more accurate, reliable, and generalizable predictive maintenance model, enhancing server health monitoring and efficiency.

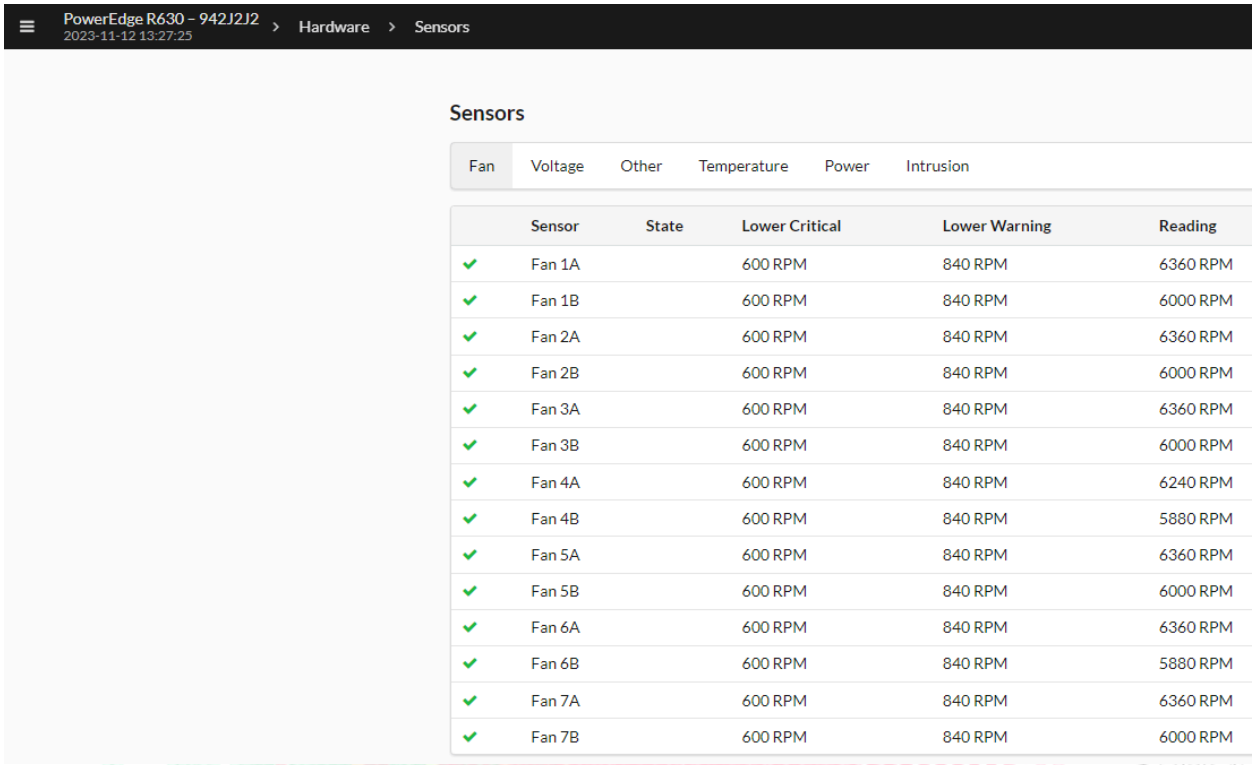
Model Usage

The models functionality was first tested within the Jupyter Notebook script where the server of serial tag “942J2J2” had sensor data informing of an issue about its HDD 1 being removed or being inside the bay incorrectly during boot up including all of its other sensor data included (CPU1 & CPU2 Temp, and average fan speed)





Above shows an image of its senor data and the time it was recording being 13:27:25, its data show a CPU1 and CPU2 temperature of 58 and 22 degrees celsius respectively and in addition to that, the servers average fan speed is revealed to be 6154.28571 rpm as shown in the next image after the calculations.



PowerEdge R630 – 942J2J2
2023-11-12 13:27:25 › Hardware › Sensors

Sensors

Fan | Voltage | Other | Temperature | Power | Intrusion

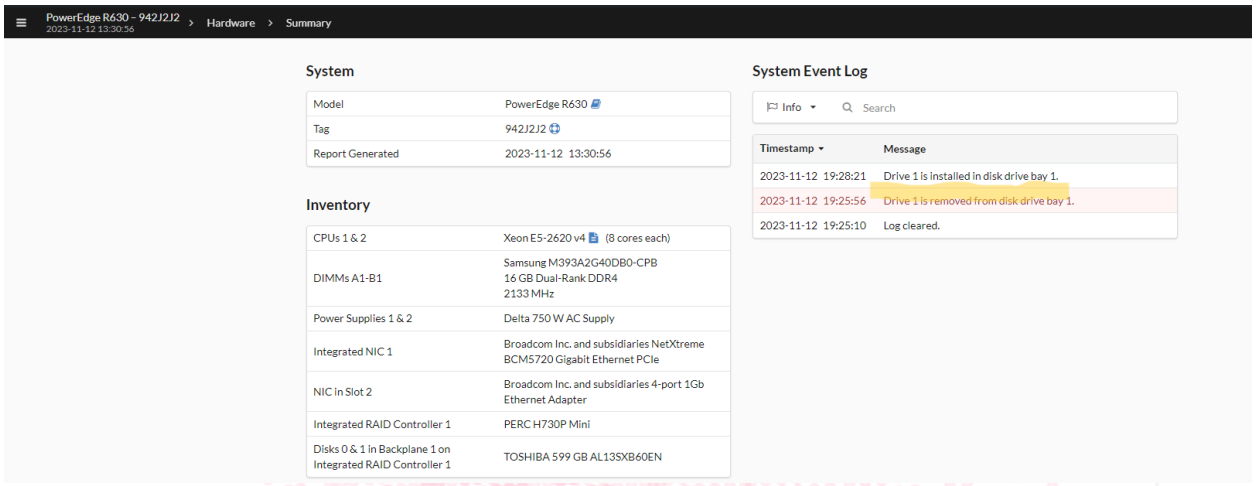
	Sensor	State	Lower Critical	Lower Warning	Reading
✔	Fan 1A		600 RPM	840 RPM	6360 RPM
✔	Fan 1B		600 RPM	840 RPM	6000 RPM
✔	Fan 2A		600 RPM	840 RPM	6360 RPM
✔	Fan 2B		600 RPM	840 RPM	6000 RPM
✔	Fan 3A		600 RPM	840 RPM	6360 RPM
✔	Fan 3B		600 RPM	840 RPM	6000 RPM
✔	Fan 4A		600 RPM	840 RPM	6240 RPM
✔	Fan 4B		600 RPM	840 RPM	5880 RPM
✔	Fan 5A		600 RPM	840 RPM	6360 RPM
✔	Fan 5B		600 RPM	840 RPM	6000 RPM
✔	Fan 6A		600 RPM	840 RPM	6360 RPM
✔	Fan 6B		600 RPM	840 RPM	5880 RPM
✔	Fan 7A		600 RPM	840 RPM	6360 RPM
✔	Fan 7B		600 RPM	840 RPM	6000 RPM

Now looking back at the script, i provided code for a user interaction with the model after its development and evaluation stage:

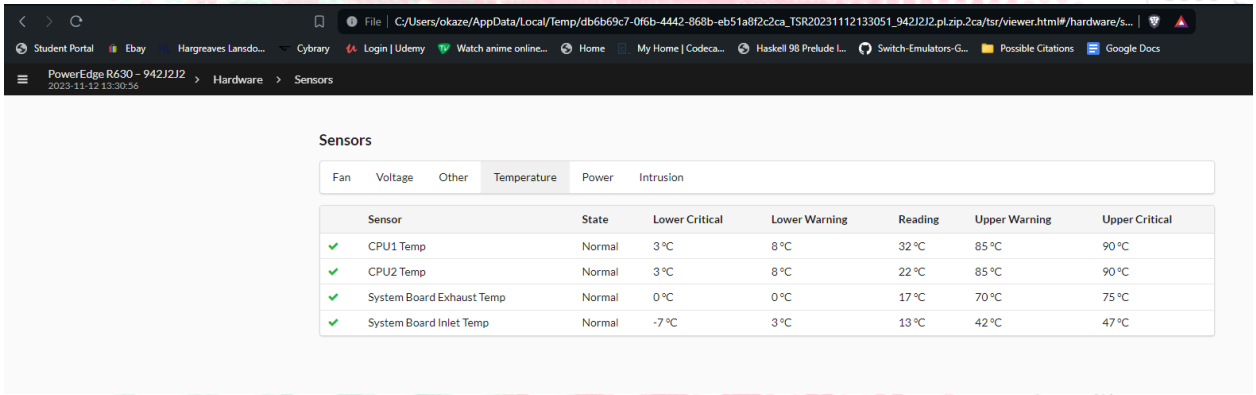
```
def predict_server_status(cpu1_temp, cpu2_temp, fan_speed, model, scaler):
    # Prepare the input data in the same format as the training data
    input_data = np.array([[cpu1_temp, cpu2_temp, fan_speed]])
    input_data_scaled = scaler.transform(input_data)

    # Make a prediction
    prediction = model.predict(input_data_scaled)
    status = 'Working' if prediction[0][0] > 0.5 else 'Not Working'
    return status

# Example usage
status = predict_server_status(58, 22, 6154.28571, model, scaler)
print("Server Status:", status)
```

```
[6]  ✓ 0.1s

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step
Server Status: Not Working
```

Here it shows that the server of Serial Tag “942J2J2” is not working. So with that, the HDD bay was filled with drive ,1 illustrated in the next screenshot →



So now we can see that it has been resolved,its current sensor readings are taken again for the model to be verified as truly working,(all screenshots are time stamped for accuracy by the iDrac firmware itself).



The image below shows all the fan readings which will give us an average fan speed of 6,188.57142 rpm with the screenshot above revealing CPU1 & CPU2 Temperature of 32 & 22 degrees celsius respectively. The top and most current log not being highlighted in red reveals that the server is functioning as intended but to be sure those values of the current sensor readings are passed through the user interaction section of the script to verify its functionality. (screenshots are provided on the next page)

PowerEdge R630 – 942J2J2 › Hardware › Sensors
2023-11-12 13:30:56

Sensors

Fan | Voltage | Other | Temperature | Power | Intrusion

	Sensor	State	Lower Critical	Lower Warning	Reading
✔	Fan 1A		600 RPM	840 RPM	6480 RPM
✔	Fan 1B		600 RPM	840 RPM	6000 RPM
✔	Fan 2A		600 RPM	840 RPM	6360 RPM
✔	Fan 2B		600 RPM	840 RPM	6000 RPM
✔	Fan 3A		600 RPM	840 RPM	6480 RPM
✔	Fan 3B		600 RPM	840 RPM	6000 RPM
✔	Fan 4A		600 RPM	840 RPM	6240 RPM
✔	Fan 4B		600 RPM	840 RPM	6000 RPM
✔	Fan 5A		600 RPM	840 RPM	6480 RPM
✔	Fan 5B		600 RPM	840 RPM	6000 RPM
✔	Fan 6A		600 RPM	840 RPM	6360 RPM
✔	Fan 6B		600 RPM	840 RPM	5880 RPM
✔	Fan 7A		600 RPM	840 RPM	6360 RPM
✔	Fan 7B		600 RPM	840 RPM	6000 RPM

```
def predict_server_status(cpu1_temp, cpu2_temp, fan_speed, model, scaler):
    # Prepare the input data in the same format as the training data
    input_data = np.array([[cpu1_temp, cpu2_temp, fan_speed]])
    input_data_scaled = scaler.transform(input_data)

    # Make a prediction
    prediction = model.predict(input_data_scaled)
    status = 'Working' if prediction[0][0] > 0.5 else 'Not Working'
    return status

# Example usage
status = predict_server_status(32, 22, 6188.57142, model, scaler)
print("Server Status:", status)
```

[17] ✓ 0.1s

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step
Server Status: Working
```

When that said sensor data of the server “942J2J2” is plugged in , the model lets us know that the server is working and that coincides with the fact that no error is at the top of the system logs meaning the model works as intended.

Real Time Model Usage (Docker)

In this section, we explore the practical application of deploying our predictive maintenance model using Docker, a platform that simplifies the deployment process by containerizing applications. This approach ensures that our model runs consistently across different computing environments. Here, we will detail the necessary components and steps involved in setting up and utilising our Dockerized predictive maintenance model to monitor server health in real-time.

Before diving into the deployment process, it's essential to understand the components that make up our Docker container. These components include:

Dockerfile:

Purpose: The Dockerfile contains all the commands a user could call on the command line to assemble an image. It automates the installation of software, Python libraries, and sets up the environment.

Content Overview: It typically starts from a base image, such as Python, and includes instructions for adding files, installing packages from the requirements.txt, and setting the command to run the Flask application.

Dockerfile

Run_Model > Dockerfile > ...

```
1   # Use a Python base image
2   FROM python:3.9
3
4   # Set the working directory in the container
5   WORKDIR /app
6
7   # Copy application files to the container
8   COPY inference_script.py requirements.txt my_model.h5 /app/
9
10  COPY scaler.pkl /app/
11
12  # Install dependencies
13  RUN pip install --no-cache-dir -r requirements.txt
14
15  # Expose the port that the Flask app runs on
16  EXPOSE 5000
17
18  # Command to run the application
19  CMD ["python", "inference_script.py"]
```

Model File (my_model.h5):

Purpose: This file contains the trained TensorFlow model. It is crucial for the Flask application to load and make predictions.

Details: The model file is built and trained in a Jupyter Notebook environment and then saved in the HDF5 file format, which allows for the storage of large quantities of numerical data.

Scaler File (scaler.pkl):

Purpose: To maintain the preprocessing consistency, the scaler file contains the scaling parameters used during the training phase to ensure that incoming data is scaled identically for accurate predictions.

Details: This file is generated from the preprocessing step of the model training process, where MinMaxScaler or any other scaling method is applied.

```
model.save('my_model.h5')

# Save the scaler
import joblib
joblib.dump(scaler, 'scaler.gz')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()`
```

Requirements File (requirements.txt):

Purpose: Lists all Python libraries and their versions needed to run the Flask application and the machine learning model within the Docker container.

Content Overview: Includes TensorFlow, Flask, NumPy, Pandas, and any other dependencies required by the application.

FinalModel.ipynb | requirements.txt

Run_Model > requirements.txt

```
1  Flask==2.0.3
2  Werkzeug==2.0.3
3  numpy
4  joblib
5  tensorflow
6  scikit-learn
```

Flask Application Script (inference_script.py):

Purpose: The core of our application, this Python script uses Flask to create a web server that can listen for incoming data, process it through the model, and return predictions.

Details: It handles web requests, loads the model and the scaler, preprocesses incoming data, performs predictions, and sends responses.

FinalModel.ipynb | requirements.txt | inference_script.py

Run_Model > inference_script.py > ...

```python
from flask import Flask, request, jsonify
import numpy as np
import joblib
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import load_model # type: ignore

# Initialize Flask app
app = Flask(__name__)

# Load the pre-trained model
model = load_model('my_model.h5')

# Load the pre-fitted scaler
scaler = joblib.load('scaler.pkl')

# Function to map numeric predictions to labels
def map_label(prediction):
    return "working" if prediction == 1 else "not working"

# Function to preprocess input data
def preprocess_input(data):
    return scaler.transform(data)

# Prediction endpoint
@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    try:
        features = np.array(data['input'])
        if features.ndim == 1:
            features = features.reshape(1, -1)

        # Preprocess features using the scaler
        features_scaled = preprocess_input(features)

        # Make prediction
        numeric_predictions = model.predict(features_scaled)

        # Convert numeric predictions to string labels
        predictions = [map_label(int(pred[0])) for pred in numeric_predictions]

        return jsonify({'predictions': predictions})

    except Exception as e:
        return jsonify({'error': str(e)}), 400

# Main entry point
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Deployment

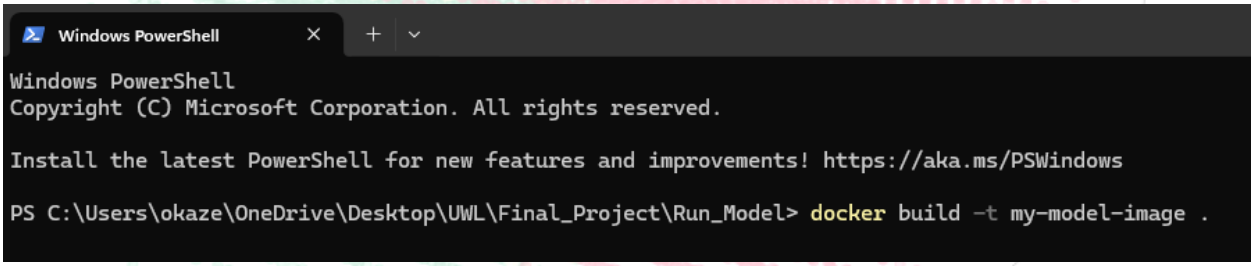
Docker is instrumental in deploying the machine learning model into production due to its ability to package the application and its dependencies into a standardised unit for software development, known as a container. This environment ensures that the model performs consistently across different computing environments, from local development machines to production servers.

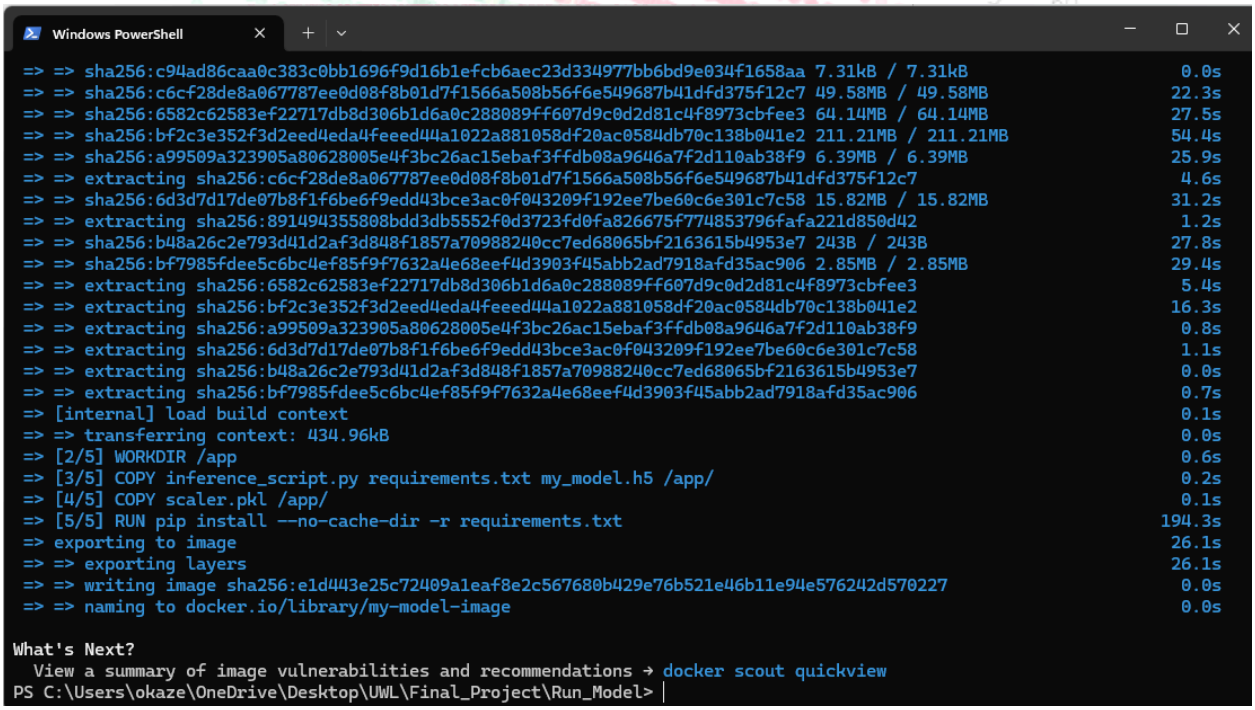
Containerization:

The Dockerfile specifies all the dependencies of the predictive maintenance model, including Python, Flask, TensorFlow, and necessary libraries listed in requirements.txt. This file defines the steps to create a Docker image.

Build the Docker image from the Dockerfile using the command provided in Instructions_cl.txt:

- docker build -t my-model-image



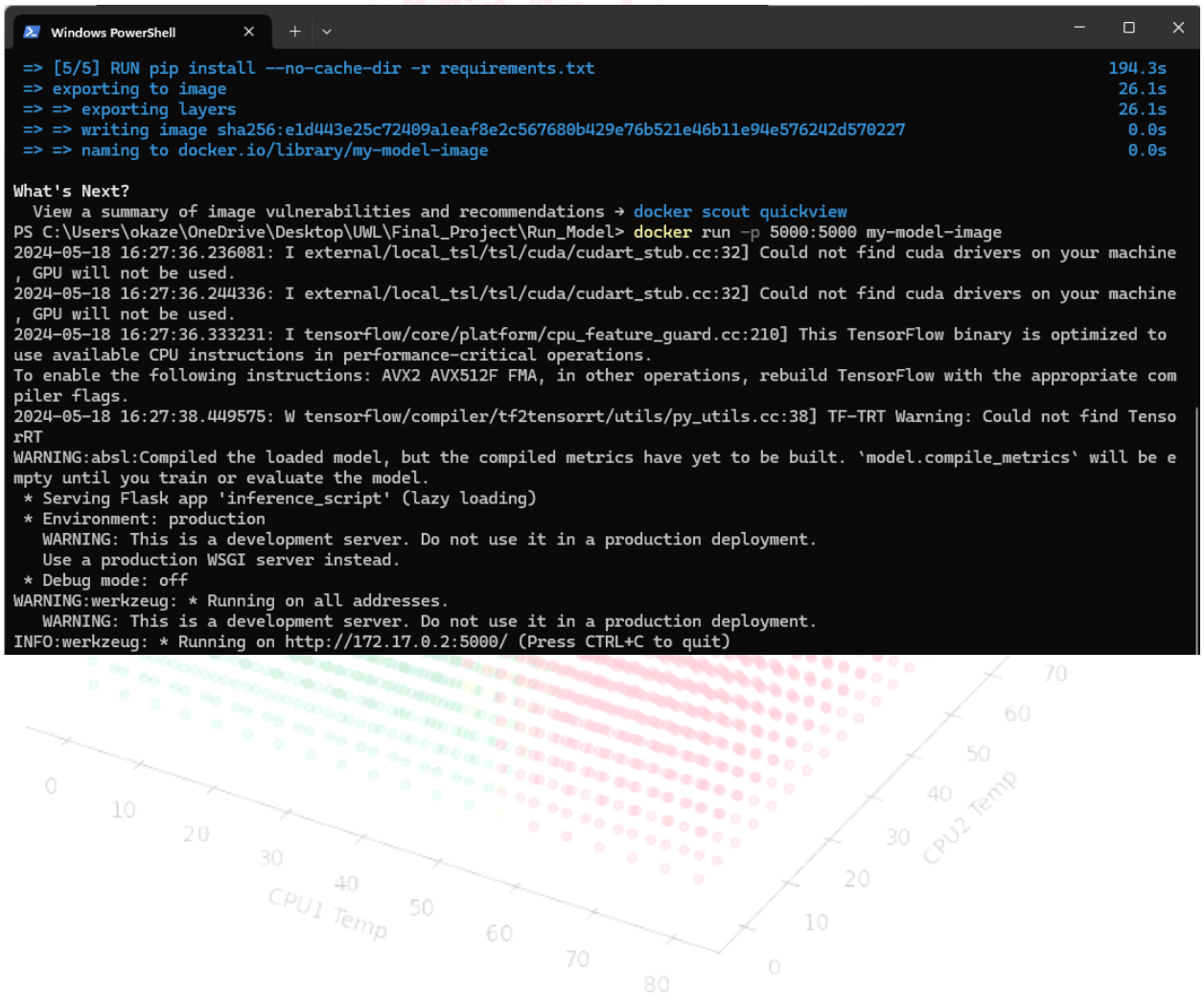


Model Serving:

Running the Docker container which hosts the Flask app defined in inference_script.py. This script loads the trained model and scaler, ready to receive data and make predictions in real-time.

Start the Docker container to run the Flask application:

- docker run -p 5000:5000 my-model-image

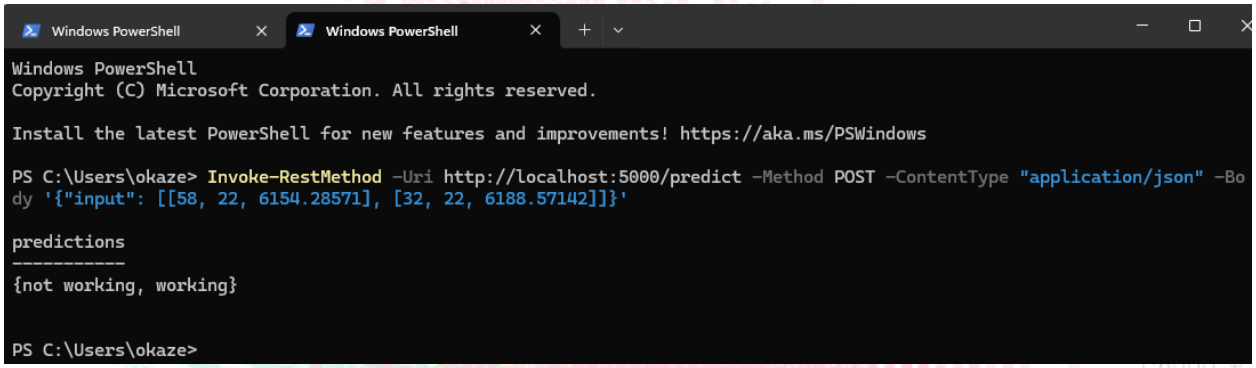


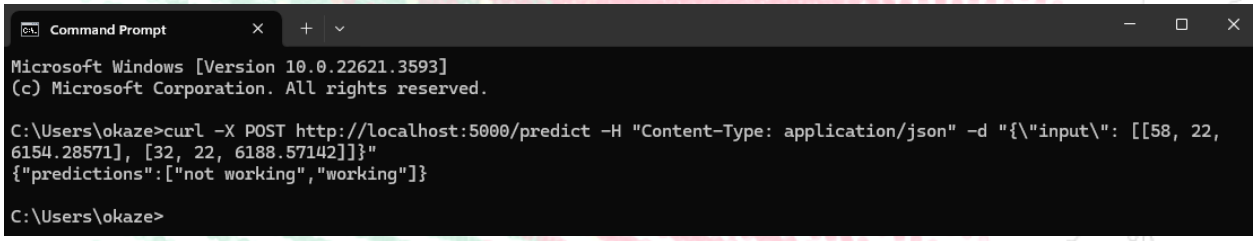
Interaction with the Deployed Model

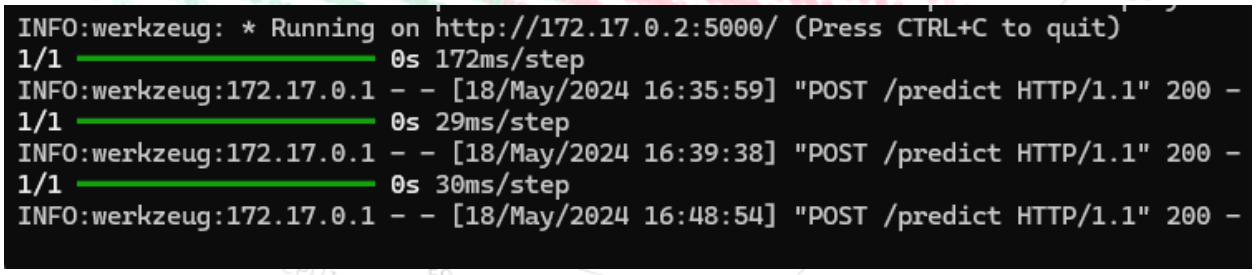
Making Predictions:

Utilising the command line or PowerShell to send POST requests to the Flask app running inside the Docker container. These requests simulate real-time data input to the model for prediction.

Using curl or Invoke-RestMethod to send JSON data representing server sensor readings, as described in Instructions_cl.txt:







The deployment of our predictive maintenance model using Docker encapsulates a robust, scalable, and efficient method for real-time server health monitoring. By containerizing the application, we ensure that the environment remains consistent across all stages of development, testing, and production, thereby mitigating potential issues arising from environment discrepancies.

The use of Docker provides several key advantages:

Portability: Once the Docker image is created, it can be run on any system that supports Docker, regardless of the underlying operating system or environment configurations.

Isolation: Docker containers operate independently of each other and the host system, which minimises conflicts between running applications.

Reproducibility: The Dockerfile specifies a clear, executable list of dependencies and setup steps, ensuring that the model can be reproduced or scaled without errors or omissions.

Efficiency: Containers can be easily started, stopped, and replicated, which is invaluable for deploying updates, scaling operations, or performing maintenance on live applications without significant downtime.

Through the Flask application housed within the Docker container, we achieve a seamless integration of our predictive maintenance model into live environments. This setup not only facilitates the monitoring of server conditions in real time but also enhances the decision-making process for maintenance interventions based on precise, data-driven predictions.

The integration and deployment process outlined in this chapter highlights the practical application of machine learning models in real-world settings, showcasing how advanced AI techniques can be effectively utilised to improve server maintenance strategies, reduce operational risks, and extend the lifespan of critical IT infrastructure. This deployment strategy, therefore, stands as a testament to the transformative potential of machine learning and AI in the field of predictive maintenance within modern IT ecosystems.

Conclusion

10000
8000
6000
4000
Average Fan Speed
0
10
20
30
40
50
60
70
80
CPU2 Temp
0
10
20
30
40
50
60
70
80
CPU1 Temp