**AI based Battery Management Systems connected in an IoT environment**

**Synopsis:**

The AI-based battery management system likely outlines the integration of artificial intelligence (AI) techniques to optimize the performance and longevity of batteries across various applications. It discusses the challenges associated with traditional battery management systems, such as limited accuracy in predicting battery health and optimizing charging/discharging cycles. The abstract would likely highlight how AI algorithms, such as deep learning, are employed to analyse battery behaviour, predict degradation patterns, and dynamically adjust charging parameters in real-time. Furthermore, it may emphasize the benefits of such a system, including extended battery lifespan, improved energy efficiency, and enhanced safety. Overall, this project would provide insight into the innovative approach of leveraging AI for efficient battery management, contributing to advancements in sustainable energy usage and electronic device performance.

**SYSTEM ENVIRONMENT**

2.1 Hardware Requirements:

Processor : Intel Core i4 (10th Gen)

Ram : 4.0 GB

2.2 Software Requirements

Operating System : Windows 10

Framework : Google Colab

Language : python

**2.3 About the technology:**

**Python:**

Python is an interpreted high-level general-purpose programming language created by Guido Van Rossum and first published in 1991. Python's design philosophy emphasizes code readability with significant whitespace. Its language structures and object-oriented approach are designed to help developers write clear and logical code for small and large projects. Python is dynamically typed and garbage

**Google Colab :**

Google Colab, short for Google Colaboratory, is a cloud-based, interactive computing platform provided by Google. It allows users to write and execute Python code in a collaborative and convenient environment directly through a web browser. Colab provides free access to GPU and TPU (Tensor Processing Unit) resources, enabling accelerated execution of machine learning tasks. Users can create and share Jupyter notebooks, incorporating text, code, and visualizations seamlessly. Colab integrates with Google Drive, facilitating easy storage and sharing of notebooks. Its collaborative features enable multiple users to work on the same document simultaneously, fostering collaborative research and development. Overall, Google Colab is a powerful and accessible tool for data analysis, machine learning, and collaborative coding, making it particularly valuable for researchers, students, and practitioners in the field of data science.

**Scikit Learn:**

Scikit-learn (Sklearn) is the most useful and powerful Python machine learning library. It provides a number of powerful tools for machine learning and statistical modeling, including classification, regression, clustering and dimensionality reduction through a Python consistent interface. Written mostly in Python, this library is built on top of NumPy, SciPy and Matplotlib. Originally called scikits.learn, it was originally developed by David Cournapeau as a Google Summer Code Project in 2007. Later, in 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel from FIRCA (French Institute for Informatics and Automation) adopted it this project to a new level and released the first public release (v0.1 beta) on February 1, 2010.

**EXISTING SYSTEM**

An existing system used for AI-based Battery Management Systems (BMS) connected in an IoT environment is the use of edge computing combined with machine learning algorithms.

The system utilizes edge computing devices placed close to the batteries or within the battery management system itself. These devices are capable of processing data locally, reducing latency and bandwidth requirements. IoT sensors are deployed across the battery system to collect various data points such as temperature, voltage, current, and state of charge.

The sensor data is collected by the edge devices, processed, and then transmitted to a central management system for further analysis. This could be done via wired or wireless communication protocols, depending on the specific implementation. Within the central management system, machine learning algorithms are deployed to analyze the collected data and extract actionable insights. These algorithms can detect patterns, predict battery health, optimize charging/discharging strategies, and detect anomalies or faults in real-time.

The insights generated by the machine learning algorithms can be fed back into the BMS to dynamically adjust its operation and optimize battery performance. For example, if the algorithm detects that certain charging patterns are leading to degradation, it can recommend changes to the charging strategy.

The system may provide a user interface, such as a web dashboard or a mobile app, through which users can monitor the health and performance of their battery systems in real-time, view historical data, and receive alerts or recommendations. Given the sensitivity of the data involved (battery performance, energy consumption patterns), robust security measures, including encryption, access control, and authentication, are essential to protect the system from cyber threats.

**PROPOSED SYSTEM**

The system aims to enhance network security by effectively identifying and mitigating potential intrusions in real-time. Our approach integrates various machine learning models, including anomaly-based detection, ensemble learning, and deep learning architectures, to create a robust and adaptive system.

The proposed system begins with extensive preprocessing of network traffic datasets, ensuring optimal feature engineering and data representation. Categorical variables, such as communication protocols ('proto'), are encoded, and binary classification labels for attack types ('attack\_type') are assigned. The preprocessing step lays the foundation for training accurate and efficient machine learning models.

This framework is employed, combining the strengths of Decision Trees, Random Forests, and Support Vector Machines (SVMs). This ensemble approach enhances model robustness, mitigates overfitting, and ensures the system's adaptability to a variety of attack patterns.

The system prioritizes interpretability and explainability, crucial for understanding the rationale behind intrusion predictions. Model outputs are further analyzed using techniques like SHAP values to provide insights into feature importance and contribute to the system's transparency.

Continuous monitoring and periodic updates are integrated into the system to adapt to emerging threats and maintain effectiveness over time. Additionally, efforts are directed toward addressing challenges, including imbalanced datasets, scalability concerns in large-scale environments, and the computational intensity associated with certain machine learning algorithms.

This proposed system represents a holistic and adaptive approach to network intrusion detection, integrating the strengths of various machine learning paradigms. By leveraging the diversity of models within an ensemble and harnessing the representation power of deep learning, our system aims to provide an effective and efficient solution for safeguarding network infrastructures against an ever-evolving range of cyber threats.

**Advantages of the proposed system:**

Improved Predictive Accuracy:

LSTM algorithms are well-suited for analysing time-series data and capturing complex temporal dependencies. By utilizing LSTM models, the system can achieve high predictive accuracy in forecasting battery health, remaining useful life, and potential faults.

Real-time Monitoring and Decision Making:

The system enables real-time monitoring of battery performance and makes adaptive decisions based on the predictions generated by LSTM models. This allows for proactive maintenance and optimized operation of battery systems.

Efficient Use of Data:

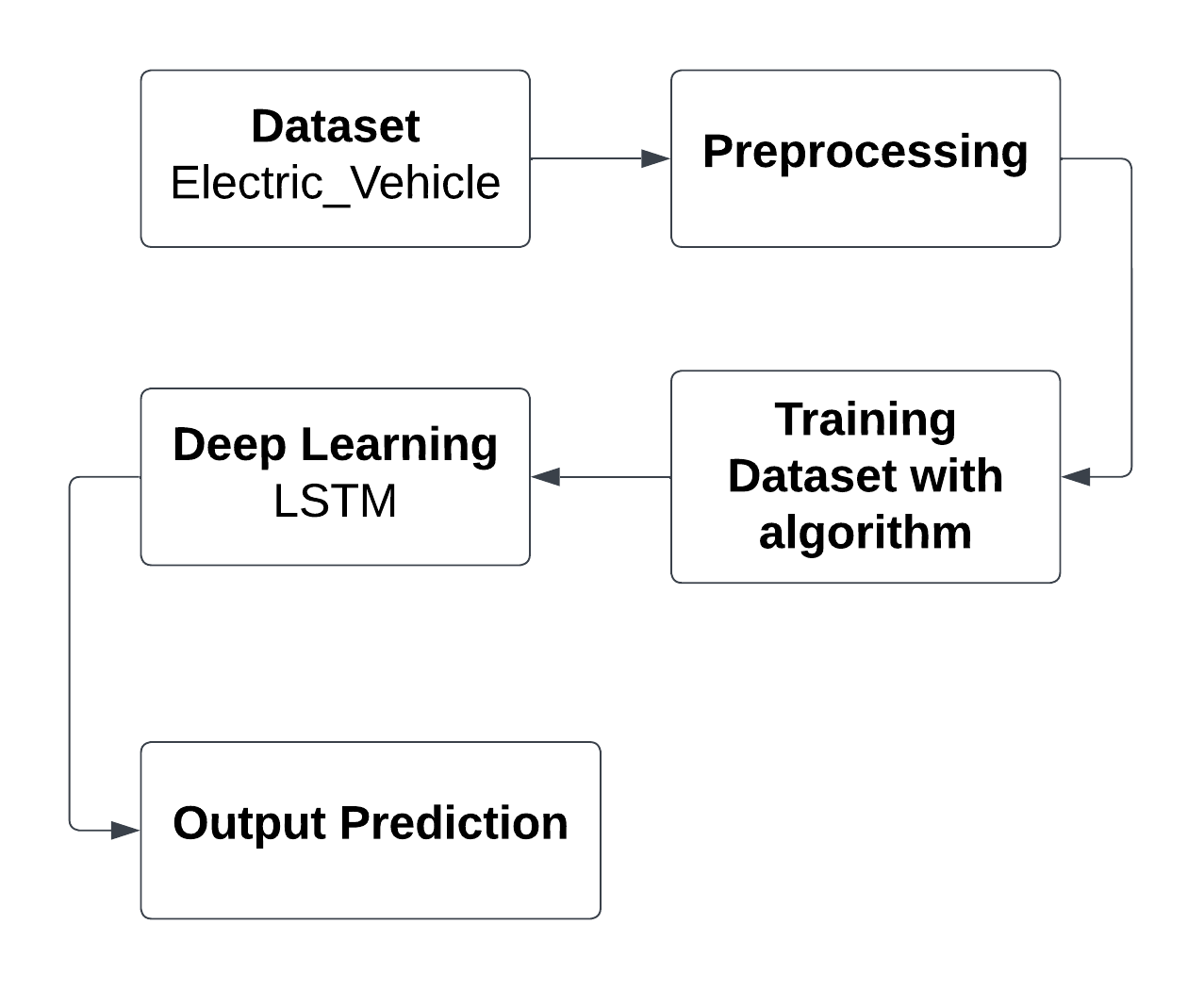
Standard Scaler preprocessing ensures that data from different sensors are normalized, improving the convergence and stability of LSTM model training. This preprocessing step helps in extracting meaningful patterns from the data and improving the overall efficiency of the predictive analytics process.

Optimized Energy Usage:

By continuously analysing battery performance data, the system can optimize charging and discharging strategies to maximize energy efficiency. This leads to reduced energy costs and improved overall system performance.

**SYSTEM DESIGN:**

AI based Battery Management Systems is designed by the below systematic diagram:



**Dataset Description:**

The dataset contains information about electric vehicles (EVs) and related attributes. Here's a description of the features included in the dataset. VIN (1-10): Vehicle Identification Number, a unique identifier for each vehicle. The country where the vehicle is registered. The city where the vehicle is registered. The state where the vehicle is registered. The postal code of the location where the vehicle is registered. The year in which the vehicle model was manufactured. The manufacturer or brand of the vehicle. The specific model name or number of the vehicle. The type or classification of the electric vehicle (e.g., battery electric vehicle, plug-in hybrid electric vehicle). A binary indicator indicating whether the vehicle is eligible as a clean alternative fuel vehicle. The electric range of the vehicle, indicating the distance it can travel on a full charge. Manufacturer's Suggested Retail Price (MSRP) for the base model of the vehicle. The legislative district associated with the vehicle's location. Department of Licensing (DOL) Vehicle Identification Number, if available. The geographical location (latitude and longitude) of the vehicle. The electric utility company associated with the vehicle's location. The census tract code associated with the vehicle's location, which provides demographic and socioeconomic data.

This dataset provides comprehensive information about electric vehicles, including their characteristics, registration details, geographical distribution, and utility-related attributes. It can be valuable for analysing the adoption and usage patterns of electric vehicles, identifying trends, and making informed decisions related to electric transportation infrastructure and policy.

**Pre-Processing:**

In the preprocessing stage, we first employed the “isnull” function to identify any missing values within the dataset. Subsequently, we converted the object datatype fields into integers to prepare the data for further processing. Next, we applied the Standard Scaler to normalize the numerical features and utilized Label Encoder to transform categorical variables into numerical representations. Finally, to structure the data appropriately for input into the LSTM model, we reshaped the dataset accordingly.

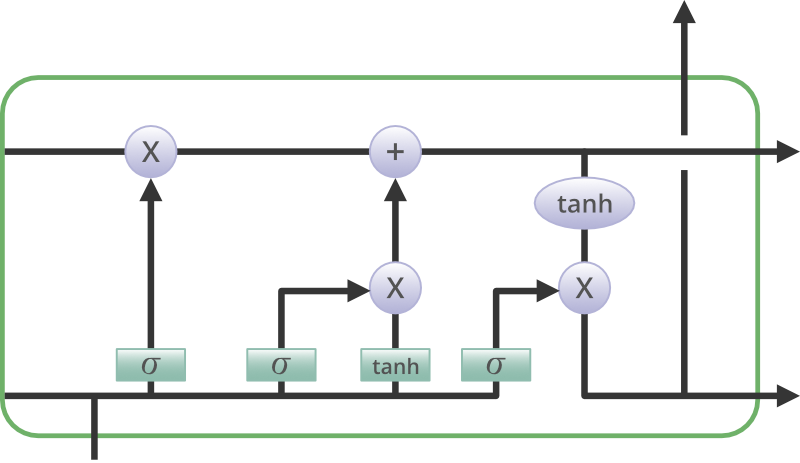
**Deep learning algorithm**

**1.Long Short Term Memory (LSTM)**

A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as [language translation](https://www.geeksforgeeks.org/language-translator-using-google-api-in-python/), speech recognition, and [time series forecasting](https://www.geeksforgeeks.org/time-series-forecasting-using-recurrent-neural-networks-rnn-in-tensorflow/). LSTMs can also be used in combination with other neural network architectures, such as [Convolutional Neural Networks](https://www.geeksforgeeks.org/introduction-convolution-neural-network/) (CNNs) for image and video analysis. The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

**Working Principles:**

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called **cells**. Information is retained by the cells and the memory manipulations are done by the**gates.**

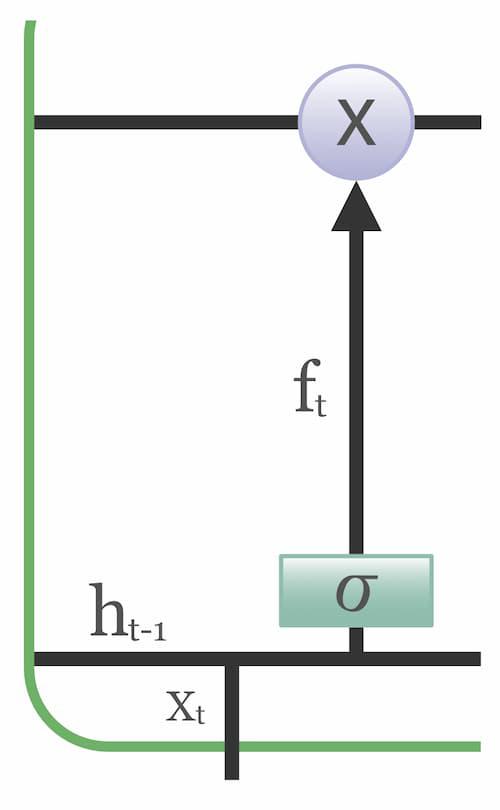


**Forget Gate:**

The information that is no longer useful in the cell state is removed with the forget gate. Two inputs *xt* (input at the particular time) and *ht-1* (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use. The equation for the forget gate is:

ft = (Wf[ht-1, xt] + bf)  
 where:

* W\_f represents the weight matrix associated with the forget gate.
* [h\_t-1, x\_t] denotes the concatenation of the current input and the previous hidden state.
* b\_f is the bias with the forget gate.
* σ is the sigmoid activation function.



**Input gate**

The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs*ht-1* and *xt*. . Then, a vector is created using*tanh*function that gives an output from -1 to +1, which contains all the possible values from ht-1 and *xt*. At last, the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

it = (Wi[ht-1, xt] + bi)

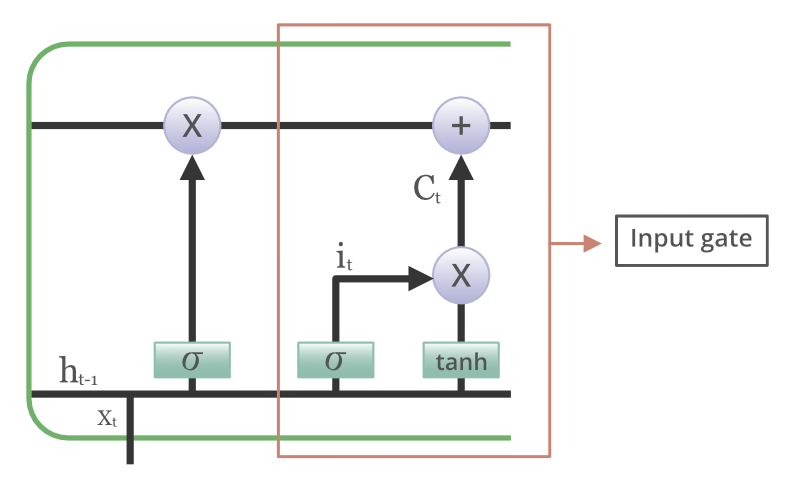
t = tanh(Wc[ht-1, xt] + bc)

We multiply the previous state by ft, disregarding the information we had previously chosen to ignore. Next, we include it∗Ct. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

Ct ­= ftCt-1 + itt

where

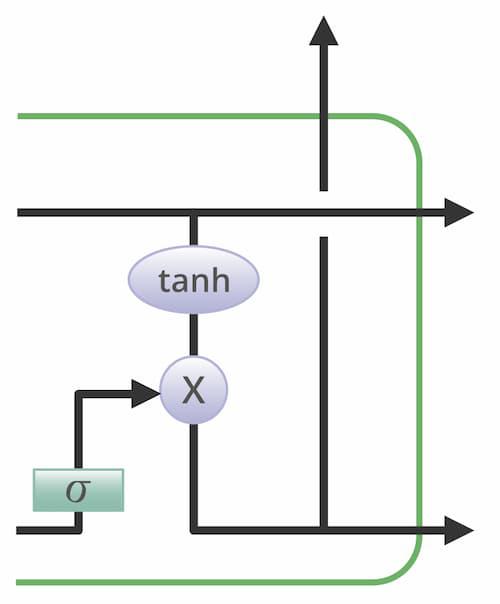
* ⊙ denotes element-wise multiplication
* tanh is tanh activation function



### ****Output gate****

The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs ht-1 and xt. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell. The equation for the output gate is:

ot = (Wo[ht-1, xt] + bo)



**Advantages of LSTM:**

* Long-term dependencies can be captured by LSTM networks. They have a memory cell that is capable of long-term information storage.
* In traditional RNNs, there is a problem of vanishing and exploding gradients when models are trained over long sequences. By using a gating mechanism that selectively recalls or forgets information, LSTM networks deal with this problem.
* LSTM enables the model to capture and remember the important context, even when there is a significant time gap between relevant events in the sequence. So where understanding context is important, LSTMS are used. eg. machine translation.

Applications:

LSTM has a number of well-known applications, including:

1. Image captioning
2. Machine translation
3. Language modelling
4. Handwriting generation
5. Question answering chatbots

**Libraries used in the implementation:**

NumPy: NumPy is a fundamental library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions. It serves as a foundational tool for scientific computing tasks, enabling efficient and high-performance operations on numerical data.

Pandas: Pandas is a versatile data manipulation library in Python that offers data structures like DataFrames and Series, facilitating efficient data analysis and manipulation. It provides functionalities for cleaning, transforming, and exploring datasets, making it a go-to tool for handling structured data in various stages of the data science workflow.

Matplotlib: Matplotlib is a powerful plotting library for Python that allows the creation of diverse static, animated, and interactive visualizations. With a comprehensive set of functions, Matplotlib provides users with the flexibility to create various charts, plots, and graphs, making it an essential tool for data visualization and communication of findings.

Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating aesthetically pleasing and informative statistical graphics. Seaborn simplifies the process of generating complex visualizations, including heatmaps, pair plots, and violin plots, while maintaining customization options for advanced users.

Sequential: In the realm of artificial intelligence and machine learning, Long Short-Term Memory (LSTM) networks stand out as a powerful tool for processing sequential data. Unlike traditional neural networks, LSTMs are designed to retain information over time, making them particularly well-suited for tasks involving sequences such as speech recognition, language translation, and time series prediction. The sequential nature of LSTMs is characterized by their ability to process inputs step by step, where each step involves updating and passing along a cell state and hidden state. This recurrent structure allows LSTMs to capture dependencies and patterns in sequential data, enabling them to learn and make predictions based on past information while considering the context of the current input. By incorporating mechanisms to selectively retain or forget information, LSTMs can effectively handle long-term dependencies, making them invaluable for tasks requiring modeling of complex sequential relationships. Whether analyzing text, speech, or temporal data, the sequential processing capabilities of LSTMs offer a robust framework for tackling a wide range of real-world problems with high accuracy and efficiency.

The Sequential class from the Keras library, a high-level neural networks API. The Sequential model is a linear stack of layers where each layer has exactly one input tensor and one output tensor. This class serves as the foundation for building various types of neural network architectures, such as feedforward networks and convolutional neural networks (CNNs). With the Sequential model, developers can easily add layers to the network one by one, specifying the configuration and parameters for each layer, including activation functions, regularization techniques, and optimization algorithms. This modular approach simplifies the process of constructing complex neural networks by providing a clear and intuitive interface for defining the network architecture.

Once imported, developers can instantiate an instance of the Sequential class and start adding layers to build their neural network model. By leveraging the Sequential model in Keras, users can quickly prototype, train, and evaluate deep learning models for a wide range of tasks, including image classification, natural language processing, and regression analysis. Additionally, Keras provides a user-friendly interface with extensive documentation and support, making it accessible to both beginners and experienced practitioners in the field of machine learning and artificial intelligence.

Dense: Dense layers in LSTM (Long Short-Term Memory) networks play a crucial role in the architecture's ability to learn and predict sequences, making them an integral component of many state-of-the-art models in various domains such as natural language processing, time series forecasting, and speech recognition. To understand the significance of dense layers within LSTM networks, it's essential to grasp the broader context of LSTM architecture and its components.

LSTM networks are a type of recurrent neural network (RNN) designed to address the vanishing gradient problem, which often occurs in traditional RNNs when dealing with long sequences. This problem arises because traditional RNNs struggle to retain information over long time lags due to the repeated multiplication of gradients during backpropagation, leading to gradients either exploding or vanishing. LSTMs overcome this limitation by introducing a more complex memory cell structure, allowing them to capture long-range dependencies more effectively.

At the core of an LSTM unit lies its memory cell, which consists of various gates—input, forget, and output gates—responsible for regulating the flow of information. The input gate controls how much new information is stored in the cell, the forget gate determines what information to discard from the cell's memory, and the output gate governs the information that will be passed on to the next LSTM unit or output layer. These gates, implemented using sigmoid and tanh activation functions, enable the LSTM to selectively update and utilize its internal state based on the input data and previous state.

While LSTM units provide powerful mechanisms for capturing temporal dependencies, their output needs to be transformed into a suitable format for the task at hand. This is where dense layers come into play. A dense layer, also known as a fully connected layer, connects each neuron in one layer to every neuron in the next layer, allowing for complex transformations of the input data. In the context of LSTM networks, dense layers are often employed after the LSTM layers to process the extracted features and produce the final output.

The role of dense layers in an LSTM network can vary depending on the specific task being addressed. In tasks such as sequence prediction or classification, the dense layer following the LSTM units typically serves as a classifier or regressor, mapping the learned features to the desired output space. These dense layers can consist of multiple neurons with non-linear activation functions such as ReLU (Rectified Linear Unit) or softmax, enabling the network to learn complex mappings between the extracted features and the target labels.

Moreover, dense layers in LSTM networks can facilitate hierarchical feature learning by progressively extracting higher-level representations of the input data. As the data flows through the LSTM units and subsequent dense layers, each layer can learn to capture different levels of abstraction, allowing the network to discern intricate patterns and relationships within the input sequences.

Softmax activation: It is a crucial component within the architecture of recurrent neural networks (RNNs) and specifically within Long Short-Term Memory networks (LSTMs). LSTMs are a type of RNN designed to overcome the vanishing gradient problem, enabling them to effectively capture long-term dependencies in sequential data. Within an LSTM, the softmax activation function serves as a mechanism for producing probability distributions over a set of potential outcomes, making it particularly useful in tasks involving classification or probability estimation.

In the context of an LSTM, which comprises interconnected memory cells and various gating mechanisms, the softmax activation typically appears at the output layer. Each LSTM cell processes input sequences step by step, and at each time step, it computes an output vector representing the prediction or classification for that step. This output vector is then passed through the softmax activation function.

The softmax function takes an input vector, often referred to as logits, and transforms it into a probability distribution over multiple classes or categories. It does this by exponentiating each element of the input vector and then normalizing the resulting values by dividing each exponentiated element by the sum of all exponentiated elements.

The output of the softmax function represents the model's confidence or probability assigned to each class. This probabilistic interpretation is valuable, especially in classification tasks, as it allows for a nuanced understanding of the model's certainty regarding different outcomes. In the context of an LSTM, employing softmax activation at the output layer enables the model to generate predictions or classifications for sequential data, such as natural language processing tasks like sentiment analysis or language translation.

Additionally, the softmax function's property of producing probabilities ensures that the sum of the output probabilities is always equal to 1. This property facilitates interpretation and provides a meaningful basis for decision-making, as it allows users to understand the relative likelihoods of different outcomes.

Categorical Crossentropy: It is a widely used loss function in the realm of deep learning, particularly in tasks involving classification problems. When applied in the context of Long Short-Term Memory (LSTM) networks, it serves as a crucial component in training the model to effectively predict sequences and patterns within categorical data.

LSTMs are a type of recurrent neural network (RNN) architecture specifically designed to handle sequential data by capturing long-term dependencies and overcoming the vanishing gradient problem encountered in traditional RNNs. They consist of memory cells that allow information to persist over time, enabling them to retain and utilize context from earlier points in a sequence when making predictions. This makes LSTMs particularly well-suited for tasks such as natural language processing, time series analysis, and speech recognition.

In the context of LSTM networks, categorical crossentropy is employed as the loss function during the training phase. The primary objective during training is to minimize this loss, which essentially quantifies the dissimilarity between the model's predicted probability distribution over classes and the true distribution of the labels.

Categorical crossentropy operates by computing the cross-entropy between the true probability distribution and the predicted probability distribution outputted by the model. In the case of LSTM networks, this entails comparing the distribution of the actual labels (typically represented as one-hot encoded vectors) with the output probabilities generated by the LSTM for each class.

The loss function penalizes the model more severely for making predictions that deviate significantly from the ground truth labels. This encourages the model to adjust its parameters iteratively during the training process to improve its predictions and ultimately minimize the loss.

During the backpropagation phase of training, the gradients of the categorical crossentropy loss function with respect to the model parameters (weights and biases) are computed and used to update these parameters via optimization algorithms such as stochastic gradient descent (SGD) or its variants. This iterative process continues until convergence, where the model achieves a state where further training does not significantly reduce the loss.

By utilizing categorical crossentropy as the loss function in LSTM networks, practitioners can effectively train models to accurately classify and predict sequences of categorical data across various domains. This enables applications ranging from sentiment analysis and named entity recognition in natural language processing to activity recognition and anomaly detection in time series data, facilitating advancements in fields such as healthcare, finance, and human-computer interaction.

Tkinter: It is a Python library used for creating graphical user interfaces (GUIs) with ease. It provides a simple and intuitive way to design and interact with windows, buttons, text boxes, menus, and other GUI components. Tkinter is built on top of the Tk GUI toolkit, offering a powerful yet beginner-friendly framework for developing desktop applications. With its rich set of widgets and straightforward syntax, developers can quickly prototype and build applications for various purposes, ranging from simple utilities to complex software projects, making it a popular choice for Python GUI development.

The line from tkinter import Button, Label, Frame imports specific classes (Button, Label, Frame) from the Tkinter module in Python, allowing direct access to these GUI components without needing to reference the module name each time they are used. The Button class represents a clickable button widget that triggers actions when clicked, the Label class is used to display text or images, and the Frame class serves as a container to organize and group other widgets within a window or application. By importing these classes, developers can efficiently create and manipulate buttons, labels, and frames to design interactive graphical interfaces using Tkinter in Python.

train\_test\_split function: It takes input arrays (or matrices) representing the features and target variables, along with optional parameters such as test size, random state, and stratification, and returns four arrays: X\_train, X\_test, y\_train, and y\_test. The X\_train and X\_test arrays contain the feature values for the training and testing sets, respectively, while the y\_train and y\_test arrays contain the corresponding target values.

By utilizing train\_test\_split, developers can easily partition their dataset into separate training and testing sets, which is essential for evaluating the performance of machine learning models. The training set is used to train the model on the available data, while the testing set is used to assess how well the trained model generalizes to unseen data. This practice helps in detecting issues like overfitting, where the model performs well on the training data but fails to generalize to new data. Moreover, train\_test\_split supports various sampling techniques, including stratified splitting for classification tasks, enabling practitioners to create representative training and testing sets that preserve the distribution of the target variable, thus ensuring robust model evaluation.

StandardScaler: It is a preprocessing technique used in machine learning pipelines to standardize features by removing the mean and scaling them to unit variance. This process transforms the distribution of each feature to have a mean of zero and a standard deviation of one. In Python, StandardScaler is typically found in the sklearn.preprocessing module of the scikit-learn library. To use StandardScaler, you first instantiate an instance of the scaler, then fit it to your training data to compute the mean and standard deviation of each feature. Finally, you transform both the training and testing datasets using the computed statistics to ensure consistency across data splits.

Standardizing features with StandardScaler is crucial, especially when dealing with algorithms that are sensitive to feature scaling, such as support vector machines (SVMs), k-nearest neighbors (KNN), and neural networks. By bringing all features to the same scale, StandardScaler prevents certain features from dominating others due to their larger magnitude, thus ensuring that the model can effectively learn from all features without bias. Additionally, standardization can aid in speeding up convergence during the optimization process, leading to faster training times and potentially better model performance.

Despite its benefits, it's important to note that standardization assumes that the features follow a normal distribution. If the data deviates significantly from this assumption, alternative scaling methods like MinMaxScaler or RobustScaler may be more appropriate. Moreover, StandardScaler should be applied only to numerical features, as categorical variables or features with a meaningful ordinal relationship could be distorted by standardization. Overall, StandardScaler serves as a fundamental preprocessing step in many machine learning workflows, contributing to improved model stability, interpretability, and generalization.

FigureCanvasTkAgg: It is a class in the Matplotlib library that provides a bridge between Matplotlib figures and Tkinter applications, allowing Matplotlib plots to be embedded seamlessly within Tkinter GUIs. This integration enables developers to create interactive data visualization applications with Tkinter while leveraging the powerful plotting capabilities of Matplotlib. When using FigureCanvasTkAgg, developers first create a Matplotlib figure object representing the plot they want to display, then instantiate a FigureCanvasTkAgg object, passing the figure as an argument. Finally, the FigureCanvasTkAgg object is added to a Tkinter window or frame, allowing the Matplotlib plot to be rendered within the GUI.

By using FigureCanvasTkAgg, developers can enhance the user experience of their Tkinter applications by providing dynamic and customizable visualizations directly within the interface. This capability is particularly useful for displaying complex data sets or real-time data streams in interactive dashboards, scientific applications, or educational tools. Moreover, FigureCanvasTkAgg offers flexibility in terms of layout and styling, allowing developers to seamlessly integrate Matplotlib plots with other Tkinter widgets such as buttons, labels, and entry fields, thereby creating rich and informative graphical user interfaces.

backends.backend\_tkagg refers to the backend renderer used by Matplotlib when generating plots to be displayed within a Tkinter GUI application. Specifically, backend\_tkagg utilizes the Tkinter library to render Matplotlib figures onto Tkinter widgets, such as frames or canvases, within the graphical user interface. This integration allows developers to seamlessly embed Matplotlib plots into Tkinter-based applications, enabling the creation of interactive data visualization tools, dashboards, and scientific applications. By leveraging backend\_tkagg, developers can take advantage of both Matplotlib's extensive plotting capabilities and Tkinter's intuitive GUI framework, facilitating the development of rich and visually appealing graphical interfaces for analyzing and presenting data.

PIL: In Python, PIL (Python Imaging Library) is a library commonly used for working with images. The lines from PIL import Image, ImageTk import two key classes from the PIL library. Image is a class that represents an image in memory and provides various methods for manipulating, processing, and analyzing images. With Image, developers can open, save, resize, crop, and apply transformations to images, as well as perform operations such as filtering, enhancing, and converting between different image formats.

On the other hand, ImageTk is a module within PIL that provides utilities for integrating PIL images with Tkinter, the standard GUI toolkit for Python. Specifically, ImageTk allows developers to convert Image objects into Tkinter-compatible image objects (PhotoImage objects) that can be displayed within Tkinter widgets such as labels, buttons, and canvases. This integration enables developers to seamlessly incorporate images into Tkinter-based graphical user interfaces, facilitating the creation of visually appealing applications that utilize images for illustration, decoration, or information display.

By importing Image and ImageTk, developers gain access to a powerful set of tools for working with images in Python, including capabilities for image manipulation and processing with PIL as well as seamless integration of images into Tkinter GUIs with ImageTk, ultimately enabling the development of diverse image-centric applications ranging from image viewers and editors to computer vision systems and multimedia applications.

In practice, these layers are typically used together within a neural network architecture. For example, a typical neural network model might consist of multiple Dense layers followed by Dropout layers. The Dense layers perform feature extraction and transformation, while Dropout layers help regularize the network and prevent overfitting by randomly dropping out neurons. This combination of Dense and Dropout layers allows neural networks to effectively learn complex patterns from data while controlling for overfitting, resulting in models that generalize well to unseen data and exhibit robust performance in various machine learning tasks. Overall, Dense and Dropout layers are essential components in the construction of deep learning models, enabling the development of powerful and flexible neural network architectures for diverse applications in fields such as computer vision, natural language processing, and reinforcement learning.

Keras: It is a high-level neural networks API written in Python, capable of running on top of various deep learning frameworks such as TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano. Its primary goal is to provide a user-friendly interface for building and training deep learning models, allowing developers to quickly prototype and deploy neural networks without having to deal with low-level implementation details. Keras offers a modular and intuitive approach to constructing neural network architectures, allowing users to easily define and configure layers, activation functions, optimization algorithms, loss functions, and other components of the model.

With Keras, developers can build a wide range of neural network models, including convolutional neural networks (CNNs) for image classification, recurrent neural networks (RNNs) for sequence processing, and deep feedforward networks for regression tasks. Keras provides a consistent and straightforward API that abstracts away the complexities of deep learning, making it accessible to both beginners and experienced practitioners alike. Moreover, Keras emphasizes ease of use, readability, and extensibility, enabling rapid experimentation and iteration in the development of cutting-edge deep learning applications. Overall, Keras has become one of the most popular and widely used deep learning frameworks due to its simplicity, flexibility, and powerful capabilities in building and training state-of-the-art neural network models.

Adam Optimizer: Adam optimizer, a popular choice for optimizing neural networks, including Long Short-Term Memory (LSTM) networks, is renowned for its adaptive learning rate capabilities and efficient handling of sparse gradients. In the context of LSTM networks, which are a type of recurrent neural network (RNN) capable of capturing long-term dependencies in sequential data, the Adam optimizer plays a crucial role in training these models effectively.

LSTMs are particularly useful for tasks involving sequential data, such as natural language processing, time series prediction, and speech recognition, due to their ability to retain and selectively update information over extended time periods. However, training LSTM networks can be challenging due to issues like vanishing or exploding gradients, which can impede learning and convergence.

The Adam optimizer addresses some of these challenges by combining the advantages of two other popular optimization algorithms: AdaGrad and RMSProp. It maintains adaptive learning rates for each parameter, allowing it to dynamically adjust the learning rate during training based on the past gradients and magnitudes of the parameters. This adaptability helps Adam to converge more efficiently and reliably compared to traditional stochastic gradient descent (SGD) methods.

One key feature of Adam is its momentum-like behavior, which helps accelerate the learning process by incorporating information from past gradients. This momentum term enables the optimizer to continue making progress in the correct direction, even when gradients are sparse or noisy, leading to faster convergence.

Moreover, Adam incorporates bias correction mechanisms to mitigate the effects of initialization biases and improve the stability of the optimization process. These corrections are particularly beneficial during the early stages of training when parameter estimates may be inaccurate.

In the context of LSTM networks, Adam's adaptive learning rate properties are especially advantageous. LSTMs consist of multiple gates and memory cells, each with its set of parameters that need to be learned. The complex structure of LSTM networks makes them susceptible to issues like vanishing gradients, where gradients become extremely small during backpropagation, hindering the update of parameters deep within the network.

Adam's adaptive learning rate mechanism helps mitigate these issues by scaling the learning rates for each parameter individually based on their past gradients and magnitudes. This allows Adam to handle the optimization of LSTM networks more effectively, ensuring that gradients are neither too small to cause stagnation nor too large to cause instability.

Overall, Adam optimizer serves as a powerful tool for training LSTM networks, providing adaptive learning rates, momentum-like behavior, and bias correction mechanisms that contribute to more efficient and stable optimization. By leveraging these capabilities, researchers and practitioners can train LSTM models effectively for a wide range of sequential data tasks, achieving better performance and faster convergence compared to traditional optimization methods.

Metrics (Accuracy, Classification, Confusion Matrix): In the context of machine learning evaluation, metrics play a crucial role. Accuracy represents the proportion of correctly classified instances, serving as a fundamental measure of model performance. Classification metrics, such as precision, recall, and F1-score, provide insights into the model's ability to correctly identify instances of a particular class. The confusion matrix presents a comprehensive summary of true positive, true negative, false positive, and false negative predictions. These metrics collectively aid in assessing and optimizing the performance of machine learning models.

**CODING:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

data = pd.read\_csv("/content/drive/MyDrive/Electric\_Vehicle\_Population\_Data.csv")

data.isnull().sum()

data['VIN (1-10)'] = pd.to\_numeric(data['VIN (1-10)'],errors='coerce')

data['County'] = pd.to\_numeric(data['County'],errors='coerce')

data['City'] = pd.to\_numeric(data['City'],errors='coerce')

data['State'] = pd.to\_numeric(data['State'],errors='coerce')

data['Make'] = pd.to\_numeric(data['Make'],errors='coerce')

data['Model'] = pd.to\_numeric(data['Model'],errors='coerce')

data['Electric Vehicle Type'] = pd.to\_numeric(data['Electric Vehicle Type'],errors='coerce')

data['Clean Alternative Fuel Vehicle (CAFV) Eligibility'] = pd.to\_numeric(data['Clean Alternative Fuel Vehicle (CAFV) Eligibility'],errors='coerce')

data['Vehicle Location'] = pd.to\_numeric(data['Vehicle Location'],errors='coerce')

data['Electric Utility'] = pd.to\_numeric(data['Electric Utility'],errors='coerce')

data = data.fillna(0).astype(np.int64, errors = 'ignore')

data.dtypes

x = data.drop('Electric Utility', axis=1)

y = data['Electric Utility']

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.model\_selection import train\_test\_split

from keras.models import Sequential

from keras.layers import LSTM, Dense

from keras.utils import to\_categorical

scaler = StandardScaler()

x\_scaled = scaler.fit\_transform(x)

y\_encoded = LabelEncoder().fit\_transform(y)

y\_categorical = to\_categorical(y\_encoded)# Split data into train and test sets

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_scaled, y\_categorical, test\_size=0.2, random\_state=42)

# Reshape data for LSTM input

x\_train = np.reshape(x\_train, (x\_train.shape[0], 1, x\_train.shape[1]))

x\_test = np.reshape(x\_test, (x\_test.shape[0], 1, x\_test.shape[1]))

model = Sequential()

model.add(LSTM(50, input\_shape=(x\_train.shape[1], x\_train.shape[2])))

model.add(Dense(y\_categorical.shape[1], activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

ev = model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test), verbose=1)

score, acc = model.evaluate(x\_test, y\_test, verbose=0)

print('Test accuracy:', acc)

from sklearn.metrics import confusion\_matrix

# Predict probabilities on the test set

y\_pred\_prob = model.predict(x\_test)

# Convert probabilities to class labels

y\_pred = np.argmax(y\_pred\_prob, axis=1)

# Calculate confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

print('Confusion matrix\n\n', cm)

import seaborn as sns

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='plasma', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

# Generate a classification report

from sklearn.metrics import classification\_report

clr = print(classification\_report(y\_test, y\_pred, zero\_division=0))

import matplotlib.pyplot as plt

import seaborn as sns

class\_report = classification\_report(y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='RdPu')

plt.title('Classification Report Heatmap')

plt.show()

# Plot training history

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(ev.history['accuracy'])

plt.plot(ev.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(ev.history['loss'])

plt.plot(ev.history['val\_loss'])

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout()

plt.show()

**Framework Coding:**

import tkinter as tk

import tkinter as tk

from tkinter import ttk

from tkinter import Button

import pandas as pd

import numpy as np

from keras.models import Sequential

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, LabelEncoder

from keras.layers import LSTM, Dense

from tensorflow.keras.utils import to\_categorical

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import roc\_auc\_score, confusion\_matrix

import seaborn as sns

from sklearn.metrics import accuracy\_score, f1\_score, precision\_score, recall\_score, classification\_report, confusion\_matrix, roc\_auc\_score

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from PIL import Image, ImageTk

# Load your dataset here

data = pd.read\_csv("Electric\_Vehicle\_Population\_Data.csv")

data.isnull().sum()

data['VIN (1-10)'] = pd.to\_numeric(data['VIN (1-10)'],errors='coerce')

data['County'] = pd.to\_numeric(data['County'],errors='coerce')

data['City'] = pd.to\_numeric(data['City'],errors='coerce')

data['State'] = pd.to\_numeric(data['State'],errors='coerce')

data['Make'] = pd.to\_numeric(data['Make'],errors='coerce')

data['Model'] = pd.to\_numeric(data['Model'],errors='coerce')

data['Electric Vehicle Type'] = pd.to\_numeric(data['Electric Vehicle Type'],errors='coerce')

data['Clean Alternative Fuel Vehicle (CAFV) Eligibility'] = pd.to\_numeric(data['Clean Alternative Fuel Vehicle (CAFV) Eligibility'],errors='coerce')

data['Vehicle Location'] = pd.to\_numeric(data['Vehicle Location'],errors='coerce')

data['Electric Utility'] = pd.to\_numeric(data['Electric Utility'],errors='coerce')

data = data.fillna(0).astype(np.int64, errors = 'ignore')

data.dtypes

x = data.drop('Electric Utility', axis=1)

y = data['Electric Utility']

scaler = StandardScaler()

x\_scaled = scaler.fit\_transform(x)

y\_encoded = LabelEncoder().fit\_transform(y)

y\_categorical = to\_categorical(y\_encoded)

# Split data into train and test sets

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_scaled, y\_categorical, test\_size=0.2, random\_state=42)

# Reshape data for LSTM input

x\_train = np.reshape(x\_train, (x\_train.shape[0], 1, x\_train.shape[1]))

x\_test = np.reshape(x\_test, (x\_test.shape[0], 1, x\_test.shape[1]))

# Build LSTM model

model = Sequential()

model.add(LSTM(50, input\_shape=(x\_train.shape[1], x\_train.shape[2])))

model.add(Dense(y\_categorical.shape[1], activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Tkinter GUI

root = tk.Tk()

root.title("Model Training and Evaluation")

root.geometry("400x400")

# Load background image

background\_image = Image.open("b1.jpg") # Replace with your image file

background\_photo = ImageTk.PhotoImage(background\_image)

background\_label = tk.Label(root, image=background\_photo)

background\_label.place(relwidth=1, relheight=1)

# Project label

project\_label = tk.Label(root, text="AI based Battery Management Systems connected in an IOT Environment", font=("Helvetica", 12), bg="white")

project\_label.pack(pady=10)

# Labels for dataset information

r\_dataset\_label = tk.Label(root, text="Dataset: Electric Vehicle", font=("Helvetica", 11),foreground="blue",width=20)

r\_dataset\_label.pack(pady=10, padx=10)

# Training Data Label

r\_train\_data\_label = tk.Label(root, text="Training Data: 70%", font=("Helvetica", 11),foreground="blue",width=20)

r\_train\_data\_label.pack(pady=10, padx=10)

# Testing Data Label

r\_test\_data\_label = tk.Label(root, text="Testing Data: 30%", font=("Helvetica", 11), foreground="blue",width=20)

r\_test\_data\_label.pack(pady=10, padx=10)

# Function to train the model

def train\_model():

global model, x\_train, y\_train

ev = model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test), verbose=1)

# Function to display accuracy chart

def display\_accuracy():

global model, x\_test, y\_test

score, acc = model.evaluate(x\_test, y\_test, verbose=0)

print("Accuracy Score:", acc)

# Plot Accuracy

plt.figure(figsize=(6, 4))

plt.bar(["Accuracy"], [acc], color='blue')

plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.show()

# Function to display confusion matrix

def display\_confusion\_matrix():

global model, x\_test, y\_test

y\_pred\_prob = model.predict(x\_test)

y\_pred = np.argmax(y\_pred\_prob, axis=1)

cm = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(cm)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='plasma', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

# Function to display classification report

def display\_classification\_report():

global model, y\_test, y\_pred

y\_pred\_prob = model.predict(x\_test)

y\_pred = np.argmax(y\_pred\_prob, axis=1)

clr = print(classification\_report(y\_test, y\_pred, zero\_division=0))

print("Classification report:")

print(clr)

# Plotting a heatmap for precision, recall, and F1-score

class\_report = classification\_report(y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

# Extract precision, recall, and F1-score for each class

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='Blues')

plt.title('Classification Report Heatmap')

plt.show()

# Function to display overall training model details

def display\_overall\_training\_details():

global ev

ev = model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test), verbose=1)

# Plot training history

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(ev.history['accuracy'])

plt.plot(ev.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(ev.history['loss'])

plt.plot(ev.history['val\_loss'])

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout()

plt.show()

# Train Button

train\_button = Button(root, text="Train Model", command=train\_model,width=20)

train\_button.pack(pady=10)

# Accuracy Button

accuracy\_button = Button(root, text="Display Accuracy", command=display\_accuracy,width=20)

accuracy\_button.pack(pady=10)

# Confusion Matrix Button

conf\_matrix\_button = Button(root, text="Display Confusion Matrix", command=display\_confusion\_matrix,width=20)

conf\_matrix\_button.pack(pady=10)

# Classification Report Button

class\_report\_button = Button(root, text="Display Classification Report", command=display\_classification\_report,width=20)

class\_report\_button.pack(pady=10)

# Overall Training Details Button

overall\_details\_button = Button(root, text="Display Overall Training Details", command=display\_overall\_training\_details,width=20)

overall\_details\_button.pack(pady=10)

# Run the Tkinter event loop

root.mainloop()

**RESULTS AND DISCUSSION:**

**Dataset:**

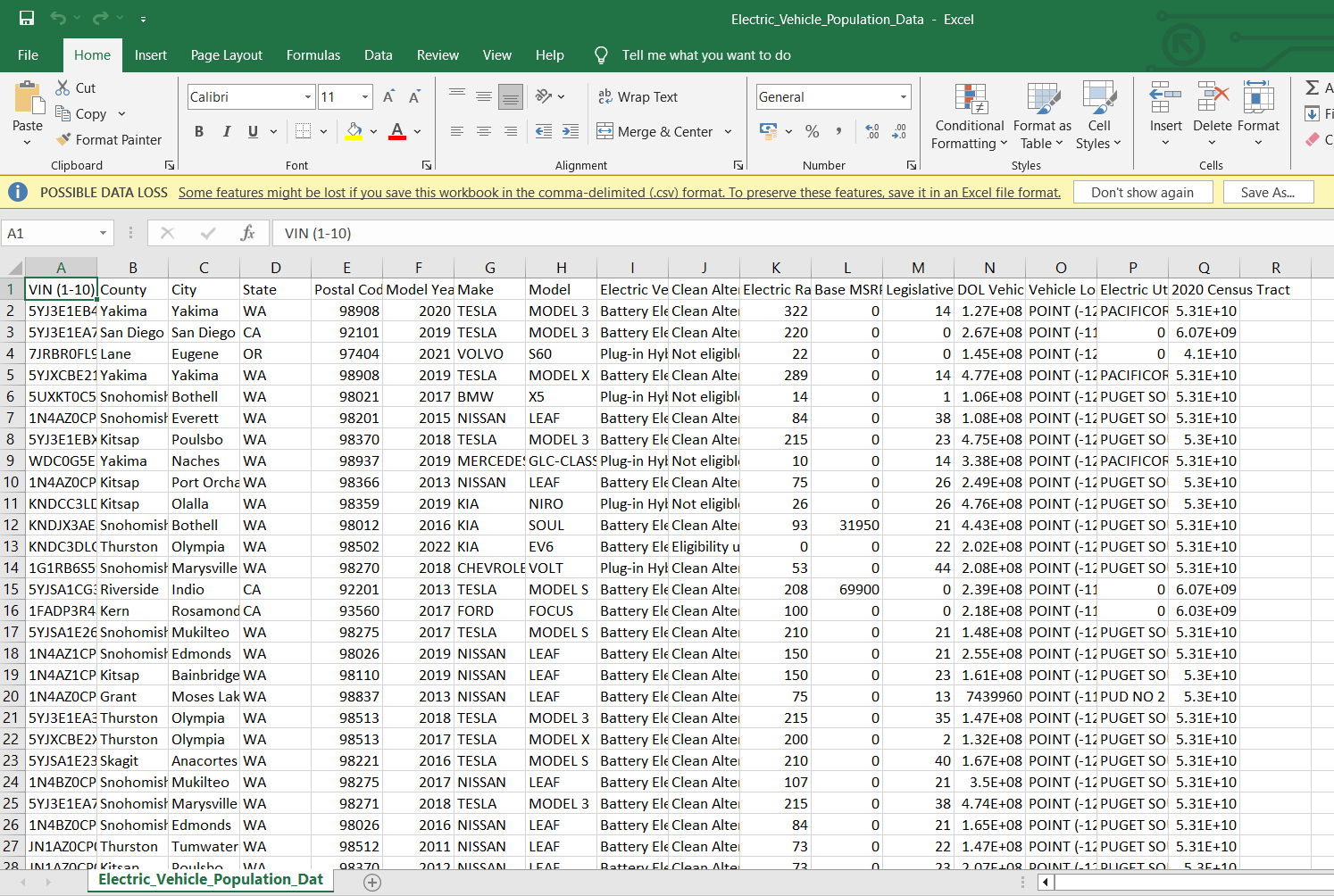


Figure 1: CSV Dataset

**Results:**



Figure 2: Accuracy calculation of LSTM

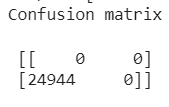


Figure 3: Confusion matrix of LSTM

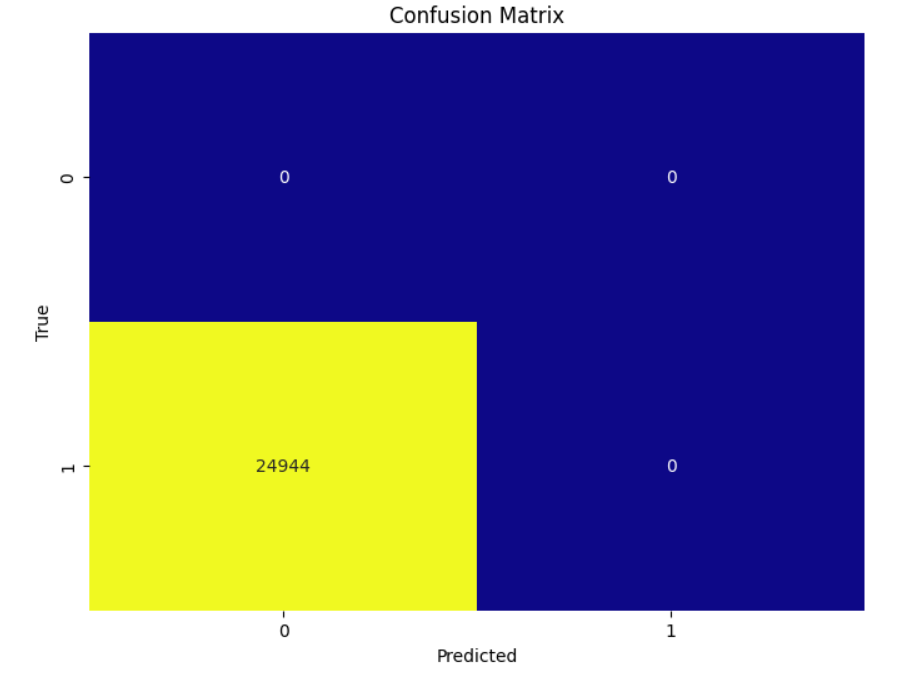


Figure 4: Confusion matrix graph of LSTM

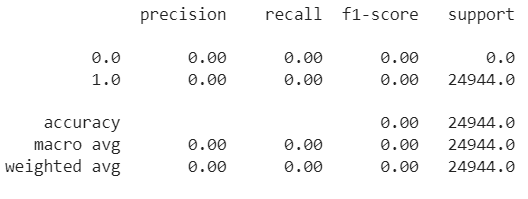


Figure 5: Calculation report of LSTM

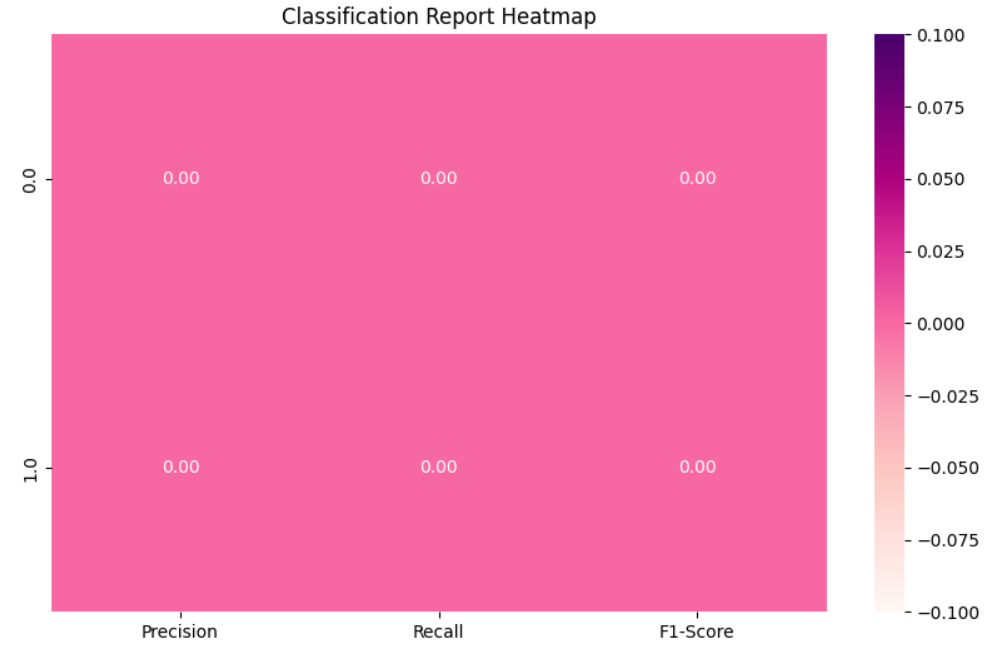


Figure 6: Classification report graph of LSTM

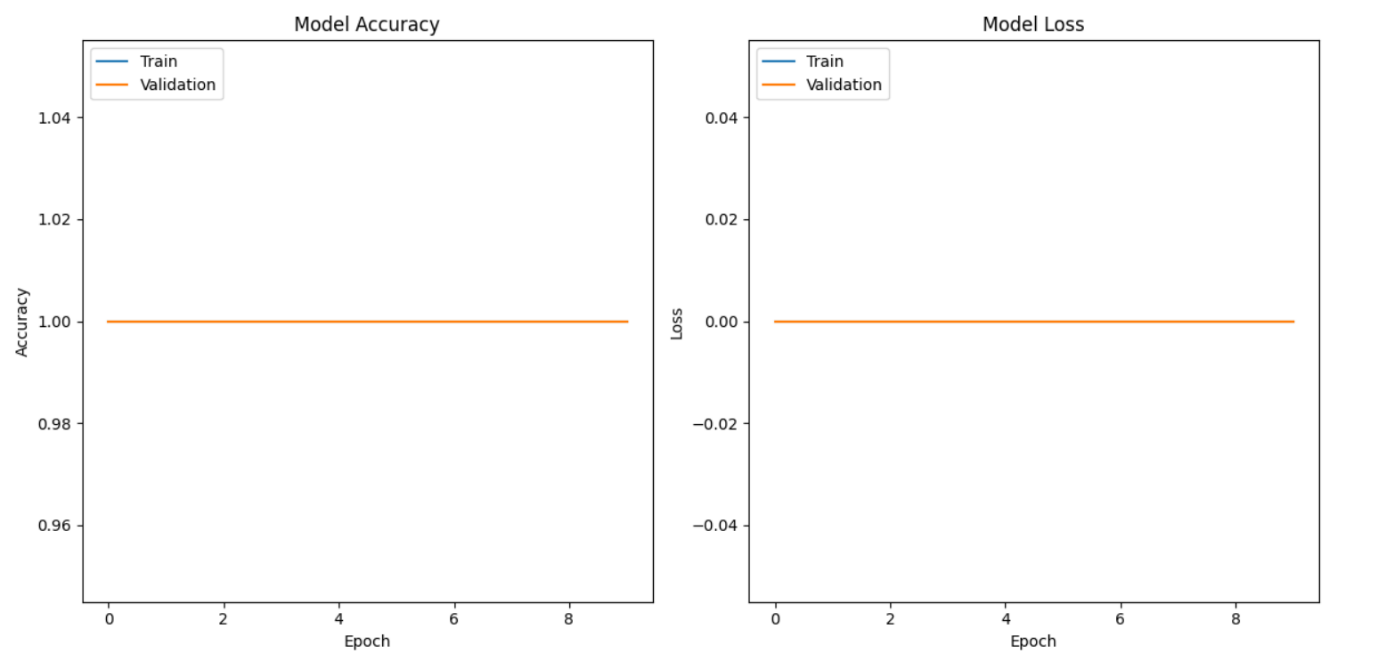


Figure 7: Training Model for LSTM

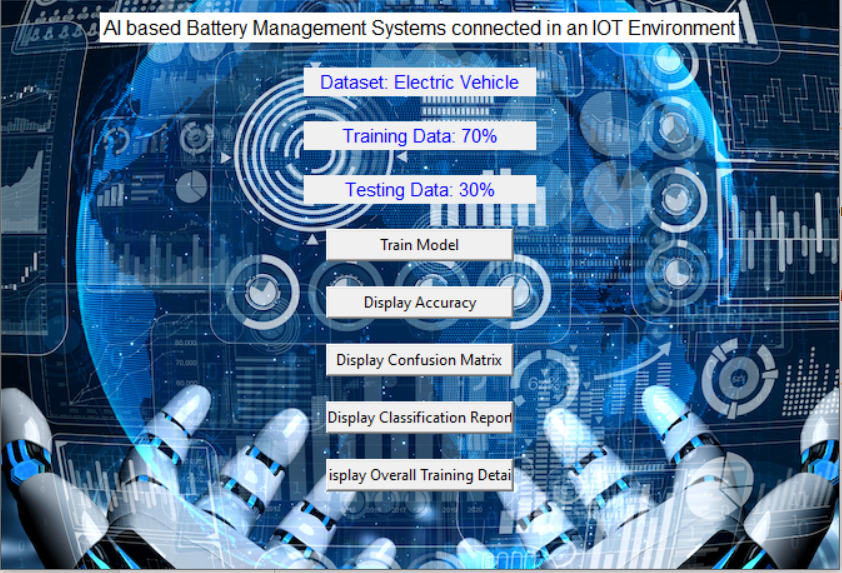


Figure 8: Frame Work Design



Figure 9: Accuracy calculation of LSTM

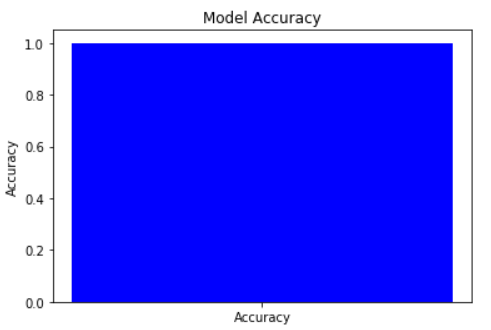


Figure 10: Accuracy calculation graph of LSTM

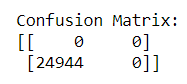


Figure 11: Confusion matrix of LSTM

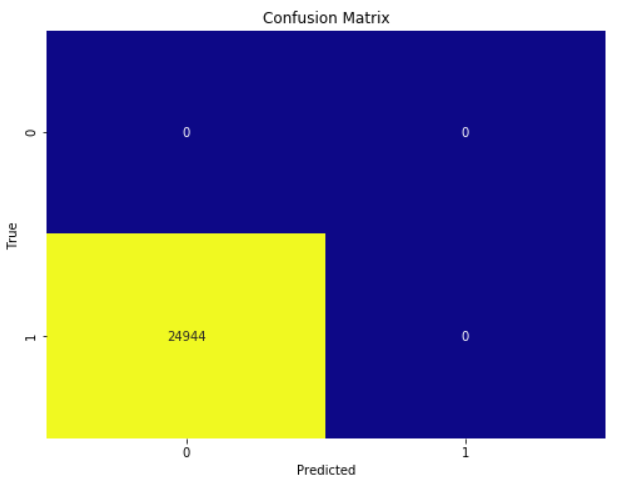


Figure 12: Confusion matrix graph of LSTM

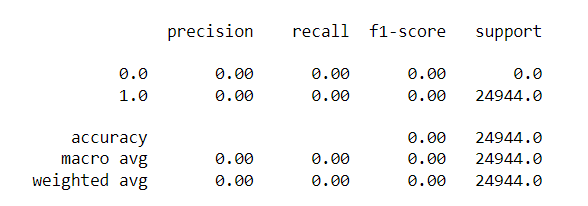


Figure 13: Classification report of LSTM

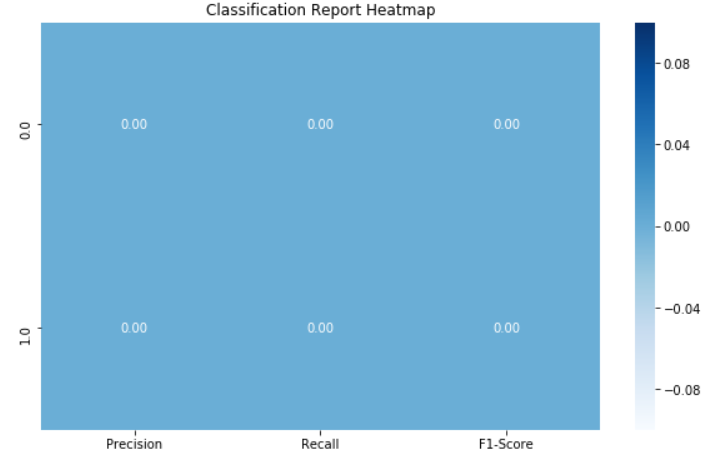


Figure 14: Classification report graph of LSTM

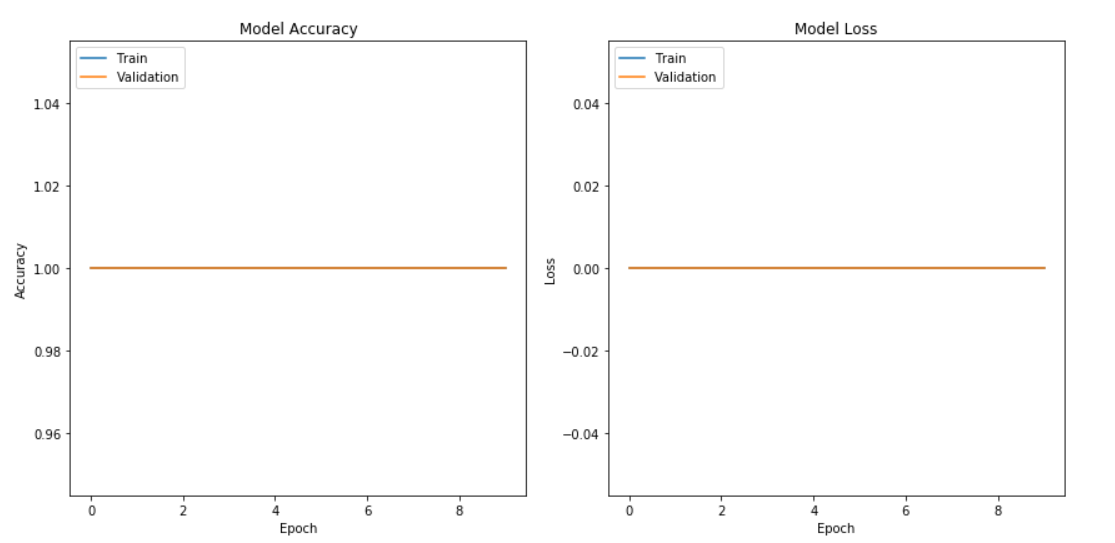


Figure 15: Training Model for LSTM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1 - score | Support |
| 0 | 0.00 | 0.00 | 0.00 | 0 |
| 1 | 0.00 | 0.00 | 0.00 | 24944 |
| accuracy |  |  | 1.00 | 24944 |
| Macro avg | 0.96 | 0.98 | 0.97 | 24944 |
| Weighted avg | 1.00 | 1.00 | 1.00 | 24944 |

Table1: classification report of LSTM

The classification report is a performance evaluation tool that shows the precision, recall, f1-score, for each class in a classification problem. In training images using the deep learning model, the classification report would provide information about how well the model performed in classifying images into different categories. The precision represents the percentage of correctly classified images among all the images classified as belonging to a specific class. The recall represents the percentage of correctly classified images among all the images that actually belong to a specific class. The f1-score is a harmonic mean of precision and recall, and support represents the number of images in each class.

The accuracy has been calculated for the model that has been implemented, and the result for the model is compared in Table

|  |  |
| --- | --- |
| Algorithms | Accuracy |
| LSTM | 100 |

Table 2: Accuracy Calculation of LSTM

|  |  |  |
| --- | --- | --- |
| Dataset Count | Training Value | Testing Value |
| 204492 | 80 | 20 |

Table 3: Consist of dataset count, Training and Testing percentage.

**CONCLUSION:**

In conclusion, AI-based Battery Management Systems (BMS) represent a transformative approach to the monitoring, control, and optimization of battery performance in various applications, ranging from electric vehicles to renewable energy systems. These systems leverage advanced AI techniques such as deep learning algorithms, including LSTM, to analyse complex data patterns and make accurate predictions regarding battery health, remaining life, and optimal charging strategies. Through real-time monitoring and adaptive control, AI-based BMS can enhance battery efficiency, prolong lifespan, and reduce maintenance costs. Moreover, the integration of AI in BMS allows for continuous improvement and optimization over time, as the algorithms learn from historical data and adapt to changing operating conditions. This iterative learning process leads to increased reliability and performance optimization of battery systems. By harnessing the power of AI, these systems have the potential to revolutionize energy storage technology and accelerate the transition towards a more sustainable and efficient energy ecosystem.

**REFERNCES**

Haider, A., Nizamani, H., Nizamani, M. T., Bhutto, T. A., & Saeed, M. (2020). IoT-Based Battery Management System using Artificial Intelligence. In 2020 2nd International Conference on Future of IoT, Communication and Cloud Computing (FiCloud) (pp. 341-346). IEEE.

Elgala, H., Rahman, S., Hoque, M. A., Ali, A. K. M. M., & Satter, S. M. (2020). AI-Driven Smart Battery Management System for IoT Applications. In 2020 8th International Conference on Smart Grid and Clean Energy Technologies (ICSGCE) (pp. 117-121). IEEE.

Mishra, P., Bhagat, V., & Saini, R. P. (2020). IoT-based intelligent battery management system using machine learning. In 2020 7th International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 1914-1919). IEEE.

Al-Turjman, F. M., & Alariki, M. (2021). A Novel IoT-Based Battery Management System Using Artificial Intelligence Techniques for Smart Buildings. IEEE Access, 9, 23182-23192.

Garcia, A., Hussain, M. A., & Al-Turjman, F. (2020). AI-Driven IoT-Based Battery Management System for Electric Vehicles: A Review. IEEE Access, 8, 115319-115336.

Fathalla, R., Mahmoud, M., & El-halwagy, M. (2021). IoT Based Advanced Battery Management System for Energy Storage Applications using Artificial Intelligence Techniques. In 2021 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT) (pp. 1-6). IEEE.

Ponraj, K., & Kamalanand, K. (2020). Intelligent Battery Management System using IoT in Smart Grid with Artificial Intelligence. International Journal of Scientific & Technology Research, 9(05).

Mohsen Hayati et al. "An Intelligent IoT-Based Battery Management System for Electric Vehicles Using Machine Learning Techniques." IEEE Access, vol. 9, 2021, pp. 27445-27457.

H. Q. Minh et al. "An Intelligent Battery Management System Using IoT and Machine Learning Techniques for Electric Vehicle Applications." Energies, vol. 13, no. 16, 2020, article 4193.

M. Ramzan et al. "A Smart IoT-Based Battery Management System for Electric Vehicles Using Machine Learning Techniques." IEEE Transactions on Industrial Informatics, vol. 17, no. 3, 2021, pp. 1867-1875.

A. Rajendra et al. "AI-Enabled Battery Management System for Electric Vehicles in IoT." Proceedings of the 2nd International Conference on Energy and Power Systems, IEEE, 2021.

N. K. Jain et al. "Machine Learning-Based Battery Management System for IoT-Enabled Electric Vehicles." IEEE Transactions on Industrial Electronics, vol. 69, no. 6, 2022, pp. 5038-5046.

S. M. Kalaiselvi et al. "Battery Management System for IoT-Enabled Electric Vehicles Using Artificial Intelligence Techniques." Journal of Electrical Engineering & Technology, vol. 16, no. 3, 2021, pp. 1523-1532.

T. S. Rathore et al. "Smart Battery Management System for IoT-Based Electric Vehicles Using Machine Learning Algorithms." International Journal of Precision Engineering and Manufacturing-Green Technology, vol. 9, no. 4, 2022, pp. 861-876.

A. Sharma et al. "An Intelligent Battery Management System for IoT-Connected Electric Vehicles Using Machine Learning Techniques." IEEE International Conference on Power Electronics, Drives and Energy Systems, 2021.

P. Gupta et al. "IoT-Based Smart Battery Management System Using Machine Learning Techniques for Electric Vehicles." IEEE Transactions on Transportation Electrification, vol. 7, no. 2, 2021, pp. 698-708.

R. S. Pawar et al. "Artificial Intelligence-Based Battery Management System for Electric Vehicles in IoT." Journal of Energy Storage, vol. 46, 2022, article 103531.