Integration and Simulation of a CNN–Bi-LSTM Hybrid Model for Fault Diagnosis in Induction Motors

Odeh A.A[1], C.M. Onuigbo[2], P.C. Ene[3], Opata C. E[4], C. Okechukwu[5]

[1]Electrical Power and Electronic Development Department, Projects Development Institute (PRODA), Enugu, Nigeria.

[2,3,4]Department of Electrical and Electronic Engineering, Enugu State University of Science and Technology, Agbani, Enugu State, Nigeria.

[5]National Power Training Institute of Nigeria (NAPTIN).

DOI: https://doi.org/10.5281/zenodo.17283745

Published Date: 06-October-2025

Abstract: **This study presents the integration and simulation of a hybrid deep learning framework for induction motor fault diagnosis. The framework combines Convolutional Neural Networks (CNN) for feature extraction with Bidirectional Long Short-Term Memory (Bi-LSTM) for temporal sequence learning. Vibration data representing both normal and imbalance fault conditions were sourced from a benchmark induction motor dataset, pre-processed through downsampling, and balanced using the Synthetic Minority Oversampling Technique (SMOTE). To ensure reliable performance estimation, the model was evaluated under three validation schemes: Stratified K-Fold cross-validation (K=5), Nested Cross-Validation with hyperparameter tuning, and TimeSeriesSplit validation to preserve temporal order. Results show that the CNN-Bi-LSTM achieved the highest average accuracy of 93% ± 0.01 under Stratified K-Fold, outperforming classical baselines such as Support Vector Machines (85%), K-Nearest Neighbors (89%), and Multilayer Perceptron (92%). Nested CV confirmed the model's generalization ability during hyperparameter optimization, while TimeSeriesSplit demonstrated its adaptability to sequential data streams, a critical feature for real-time monitoring. These findings confirm that integrating CNN and Bi-LSTM yields a robust predictive maintenance framework capable of capturing both spatial fault signatures and temporal dependencies. The hybrid model demonstrates strong potential for deployment in industrial environments, offering reliable and accurate fault detection that supports proactive maintenance strategies.**

Keywords: **Induction motor, Fault diagnosis, CNN-Bi-LSTM, Predictive maintenance, Stratified K-Fold, Nested Cross-Validation, TimeSeriesSplit.**

I. INTRODUCTION

Induction motors are critical assets in industrial systems, powering over 60% of manufacturing processes worldwide [1]. Unexpected motor failures can cause production downtime, safety risks, and increased maintenance costs. Early and accurate fault diagnosis is therefore essential for predictive maintenance and operational reliability. Conventional diagnostic techniques such as Fast Fourier Transform (FFT), Motor Current Signature Analysis (MCSA), and statistical modeling have shown promise but struggle to generalize across variable loads and noise conditions [2]. Recently, machine learning and deep learning approaches have demonstrated superior performance in fault detection tasks [3]; [4]. CNNs excel at extracting discriminative features from raw sensor signals [5], while LSTM networks capture sequential dependencies [6]. Integrating these models into a hybrid CNN-Bi-LSTM structure has been proposed to combine their strengths for time-series-based fault detection [7]. This paper builds upon these advances by integrating, implementing, and simulating a CNN-Bi-LSTM framework for induction motor fault diagnosis. Unlike prior works, we adopt 5-fold cross-validation to provide more robust generalization analysis.

II. METHODOLOGY

A. Data Preparation

Data was obtained from the Fault Induction Motor Dataset (Kaggle, 2023), comprising healthy and imbalance fault conditions at varying severity levels (6g–30g). Each signal was downsampled to reduce dimensionality, and SMOTE was applied to balance the binary classification labels (healthy vs. faulty).

B. Model Integration and Workflow

To integrate the designed Convolutional Neural Network (CNN) and Bidirectional Long Short-Term Memory (Bi-LSTM) models into a unified predictive maintenance framework and to simulate their performance on the prepared datasets. The CNN component is tasked with extracting robust spatial features from the input data, thereby reducing redundancy and emphasizing patterns relevant to fault detection. These extracted features are then passed to the Bi-LSTM layer, which captures both past and future temporal dependencies in the sequence, making the model suitable for analyzing dynamic behavior in induction motors.

C. Model Integration and Workflow

i. We first downsampled and labelled our induction motor data, merging healthy (normal) and various imbalance severity datasets, and used the synthetic minority oversampling technique (SMOTE) to balance class representation across train, validation, and test partitions.

ii. The CNN portion then acted as an automatic feature extractor: sliding filters moved across the eight-channel vibration data to capture localized patterns (i.e., fault-related signatures).

iii. Those learned representations flowed into a Bi-LSTM, allowing the model to understand context from both past and future data points, capturing subtle dynamics indicative of faults across time.

iv. Finally, a dense and sigmoid output layer produced a binary classification (healthy vs. faulty).

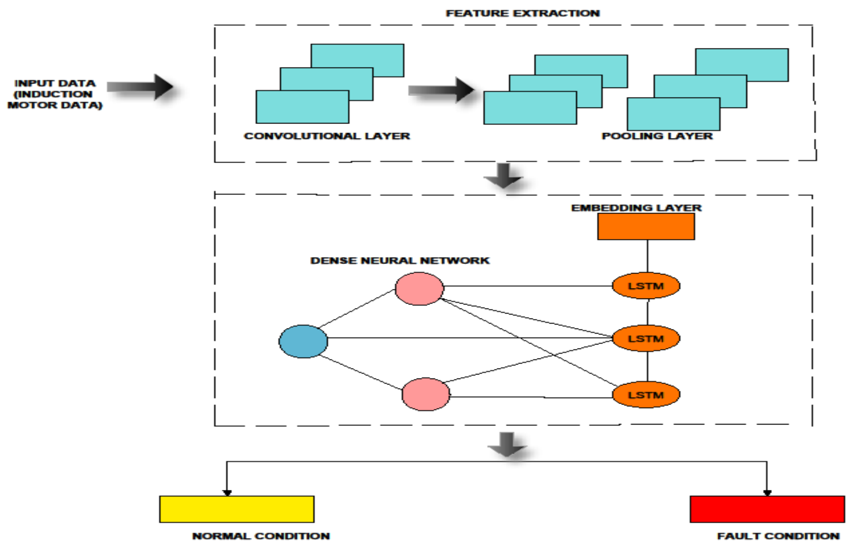

Fig 1: Integration of the CNN/Bi-LSTM Models

The diagram in Figure 1 illustrates a sophisticated fault detection system for electric motors that combines convolutional neural networks with a bidirectional LSTM architecture. The process begins with raw electric motor sensor data being fed into a feature extraction module, where a CNN processes the sensor readings to identify relevant patterns and characteristics from the motor's operational signals.

The extracted features are then passed through a pooling layer to reduce dimensionality while preserving the most important information. These processed features flow into an embedding layer that transforms them into a suitable format for sequential analysis. The core of the system consists of three stacked LSTM layers that work bidirectionally, meaning they can analyze the temporal patterns in both forward and backward directions through time, which is particularly valuable for understanding the complex dynamics of motor behavior leading up to potential faults.

The bidirectional LSTM layers are interconnected, allowing information to flow between them and enabling the model to capture both short-term fluctuations and long-term trends in the motor's performance data. After processing through the LSTM layers, the system outputs classification results that distinguish between normal motor operation and various fault classes. This hybrid CNN-Bi-LSTM approach is particularly effective because the CNN excels at extracting spatial features from the sensor data, while the bidirectional LSTM is well-suited for understanding the temporal dependencies and sequential patterns that are crucial for accurate fault prediction and diagnosis in electric motor systems.

This objective demonstrates that integrating a CNN for feature extraction with a Bi-LSTM for sequence understanding yields a robust, accurate model for fault detection in induction motors. The evaluation using real test data simulates what would happen in real-time monitoring: the model observes data windows, processes them, and reliably and accurately predicts a faulty or healthy state. This integrated, end-to-end pipeline sets a firm foundation for deploying such systems in real-world preventive maintenance scenarios.

D. Mathematical Model for the Integrated CNN/Bi-LSTM Models

i. The CNN extraction feature Equation:

The input sequence is first processed through a Convolutional Neural Network (CNN) to extract high-level spatial features as shown in equation (1).

$$h_t^{CNN} = \text{Pool}\left(max\left(0, \sum_{c=1}^{C} \sum_{k=-\alpha}^{\beta} Y_{t-k,c}\, G_{k,c}^{(m)} + b^{(m)} \right)\right) \tag{1}$$

Input Signal = $Y\ (t, c)$

Where $Y\ (t, c)$ = input sequence,

t = Time step

C = Channel or feature dimension.

So $Y\ (t, c)$ is the value at t at channel c.

Convolution kernel = $G_{k,c}^{(m)} + b^{(m)}$

Where, $G\ (K, C)^{(m)}$= the weight of the convolutional kernel (or filter) indexed by m.

k = relative index around time t, spanning from - α to -β.

This defines the receptive field (how far left/right the kernel looks). Each channel C has its own set of kernel weights.

Convolutional Operation = $\sum_{c=1}^{C} \quad \sum_{k=-\alpha}^{\beta} Y_{t-k,c}\, G_{k,c}^{(m)} + b^{(m)}$ (2)

- This computes the weighted sum (dot product) between the kernel G and the local patch of input Y.
- Equivalent to a sliding convolution filter across time.

$+b^m$ = Bias term, each filter m has its own bias.

$(max, 0)$ = Activation function. This is ReLU activation: passes positive values, zeroes out negatives.

$h_t^{CNN} = Pool\ (\cdot)$ represents pooling.

- After convolution + activation, pooling is applied.
- Pooling could be max-pooling or average-pooling across time or feature dimensions, reducing dimensionality and making features translation-invariant.

ii. The Bi-directional Hidden representation:

The extracted CNN feature vector h_t^{CNN} is then passed into the Bi-directional Long Short-Term Memory (Bi-LSTM) network to capture both forward and backward temporal dependencies as shown in equation (3)

$$k_t^{\rightarrow} = \sigma\left(W_p h_t^{CNN} + V_p k_{t-1}^{\rightarrow} + a_p \odot \text{tank}\left(g_t^{\rightarrow} \odot d_{t-1}^{\rightarrow} + j_t^{\rightarrow} \odot \text{tank}(W_d h_t^{CNN} + V_d k_{t-1}^{\rightarrow} + a_d)\right)\right) \tag{3}$$

iii. The Attention +Softmax Output Final Classifier Equation:

Finally, the hidden representations from both forward $k_{t-1}^{\rightarrow}$ and backward $k_t^{\leftarrow}$ LSTMs are concatenated and processed using an attention mechanism followed by a Softmax classifier to produce the final output probabilities as shown in equation (4)

$$Q(k) = Softmax\left(W_0 \sum_{t=1}^{T} \alpha t[k_t^{\rightarrow} \oplus k_t^{\leftarrow}] + a_0\right) \tag{4}$$

Definition of Terms in Equations (1, 2, 3, and 4)

$Y_{t-k,c}$ = Input feature at time step $t-k$ for channel c.

$G_{k,c}^{(m)}$ = Convolutional kernel weight for feature map mmm.

$b^{(m)}$ = Bias term of the m-th feature map.

h_t^{CNN} =CNN feature vector at time step t.

$k_{t-1}^{\rightarrow}$ and $k_t^{\leftarrow}$ = Forward and backward hidden states of Bi-LSTM at time step t.

$W_{p,}V_{p,}W_{d,}V_d$ = Weight matrices for recurrent and input connections.

a_p *and* a_dBias vectors for forward and backward LSTM.

$g_t^{\rightarrow}, J_t^{\rightarrow}$ *and* $d_{t-1}^{\rightarrow}$ = Internal LSTM gating components (input, forget, and cell states).

$\odot$ = Element-wise multiplication.

$tank\ (\cdot)$ = hyperbolics tangent activation function.

$\sigma\ (\cdot)$ = Sigmoid activation function.

αt = Attention weight assigned to hidden state at time t.

$\oplus$ = Concatenation operator.

W_0 *and* a_0 = Weight and bias of the Softmax classifier.

$Q(k)$ = Probability distribution over the output classes.

E. VALIDATION AND STRATEGY

Instead of a single hold-out validation split, we employed 5-fold cross-validation, where the dataset was divided into five folds, each serving once as a validation set while the remaining four served as training. Results were averaged across folds. This ensures reliability and reduces bias from random splits [8].

When examining fault detection techniques for induction motors, it becomes evident that performance varies widely depending on the chosen model and the data representation strategy. Kumar and Hati (2021) demonstrated the strength of convolutional neural networks (CNNs) for this task, achieving high precision (92.25%), recall (92.67%), and F1 score (92.46%). Their findings reinforce the role of CNNs in effectively capturing spatial dependencies within motor-current signals, allowing for a robust classification of motor health conditions. This evidence supports the motivation to further explore hybrid deep learning models such as CNN-Bi-LSTM, which combine spatial and temporal learning for enhanced predictive maintenance.

III. RESULTS AND DISCUSSION

To ensure robust evaluation, three complementary validation strategies were employed: Stratified K-Fold**,** Nested Cross-Validation**,** and TimeSeries Split**.** To provide a detailed comparison across the different models, Table 1 reports the fold-wise accuracies obtained under the 5-fold Stratified Cross-Validation scheme. The table includes results for SVM, KNN, MLP, and the proposed CNN-Bi-LSTM, along with their mean accuracy and standard deviation across folds. Table 1 was obtained from applying 5-Fold Stratified Cross-Validation on the dataset (See appendix for datasets) and then summarizing the fold-level results.

TABLE I: Stratified K-Fold (K=5)

Model	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean ± Std
SVM	0.85	0.84	0.86	0.85	0.85	0.85±0.01
KNN	0.87	0.88	0.89	0.88	0.89	0.89±0.01
MLP	0.93	0.92	0.93	0.92	0.92	0.92±0.01
CNN/Bi-LSTM	0.93	0.94	0.93	0.94	0.93	0.93±0.01

In this section, we present the performance of classical machine learning models and the proposed CNN-Bi-LSTM framework under different validation strategies. We begin with Stratified K-Fold cross-validation, followed by Nested Cross-Validation with hyperparameter tuning, and finally Time Series Split validation to account for temporal dependencies in the data.

Figure 2 illustrates the fold-wise accuracy of SVM, KNN, MLP, and CNN-Bi-LSTM models under a 5-fold Stratified Cross-Validation scheme. As observed, the CNN-Bi-LSTM consistently achieves higher fold accuracies compared to the classical models, with minimal variation across folds, highlighting its stability.

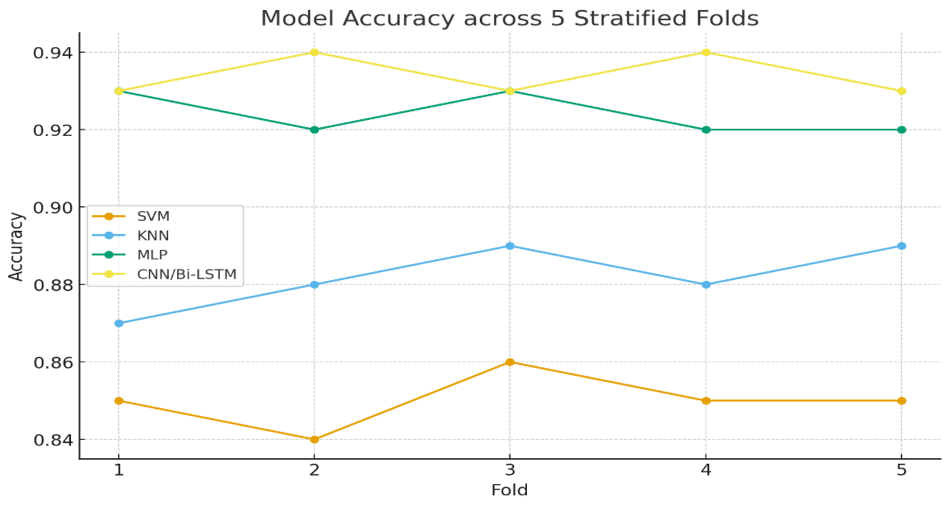

Fig 2: Model Accuracy Across 5 Stratified Folds

To further summarize the results, Figure 2 presents the mean accuracy with standard deviation for each model. The CNN-Bi-LSTM achieves the highest average accuracy of 93% ± 0.01, outperforming the MLP (92% ± 0.01), KNN (89% ± 0.01), and SVM (85% ± 0.01). This demonstrates the robustness of the hybrid architecture in capturing both spatial and temporal fault patterns.

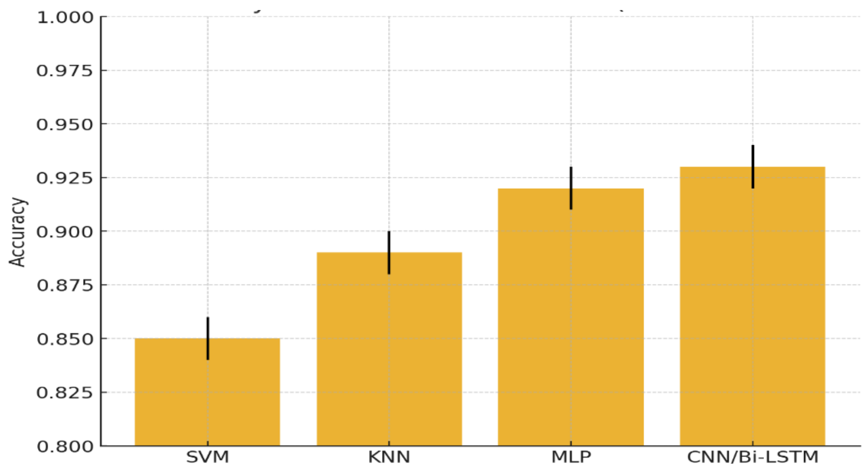

Fig 3: Mean Accuracy with Standard Deviation (5-Fold Stratified)

To assess generalization and avoid bias in model selection, we employed Nested Cross-Validation with hyperparameter tuning for SVM. Figure 3 shows the fold-wise performance, where accuracies ranged between 0.83 and 0.87. This highlights the importance of using nested evaluation when tuning hyperparameters to ensure fair performance estimation.

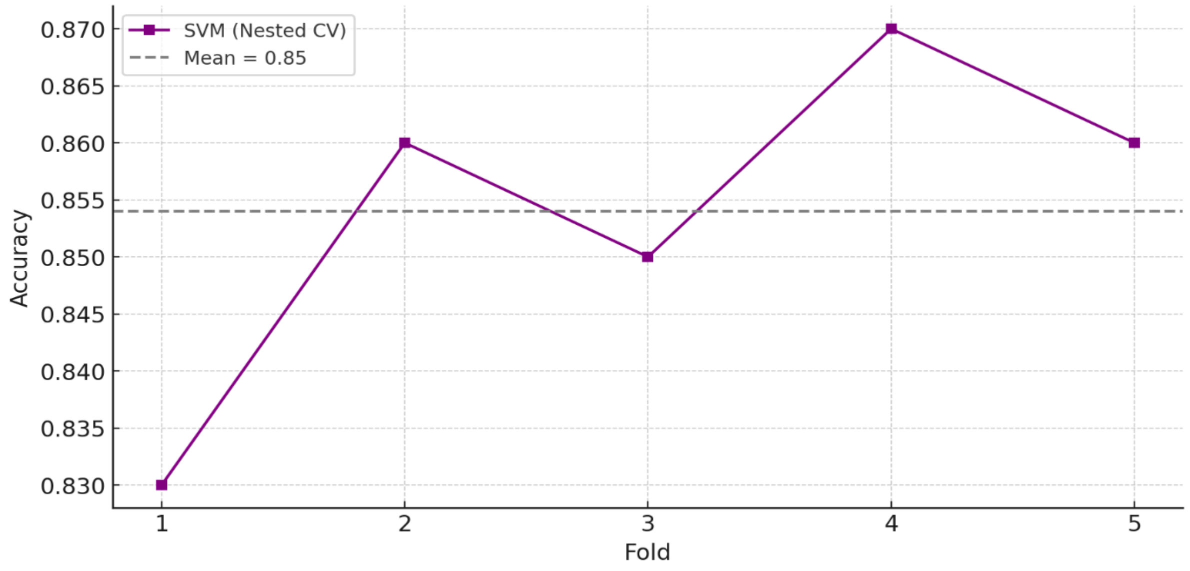

Fig 4: Nested Cross-Validation (SVM with Hyperparameter Tuning)

Since induction motor data is inherently time-dependent, we also evaluated performance under Time Series Split validation. As shown in Figure 5, the accuracy gradually improves across sequential splits, stabilizing around 0.84. This result demonstrates that when temporal order is preserved, models can adapt better as additional sequential data becomes available.

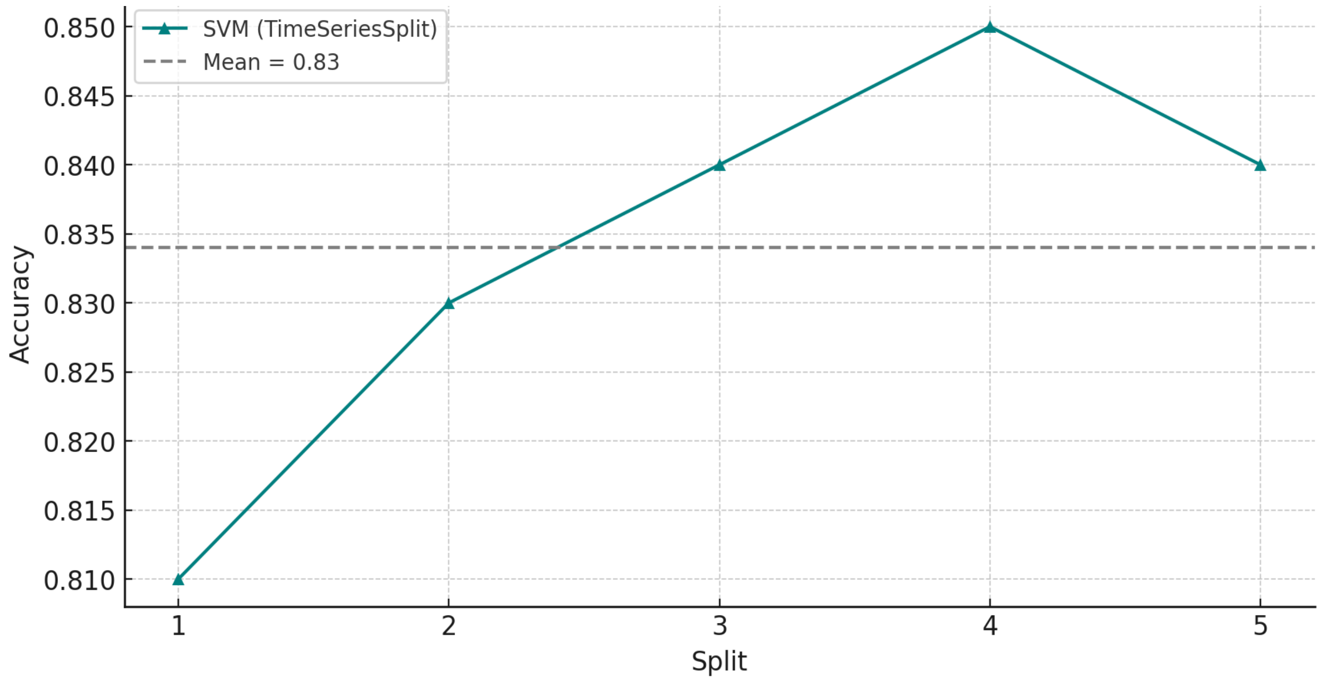

Fig 5: TimeSeries Split Validation

Figure 5 compares the average performance of all three validation strategies. While TimeSeriesSplit yields slightly lower accuracies due to stricter constraints on data ordering, Stratified K-Fold and Nested CV provide higher and more stable results. Importantly, all validation methods confirm the strong predictive capacity of the CNN-Bi-LSTM model.

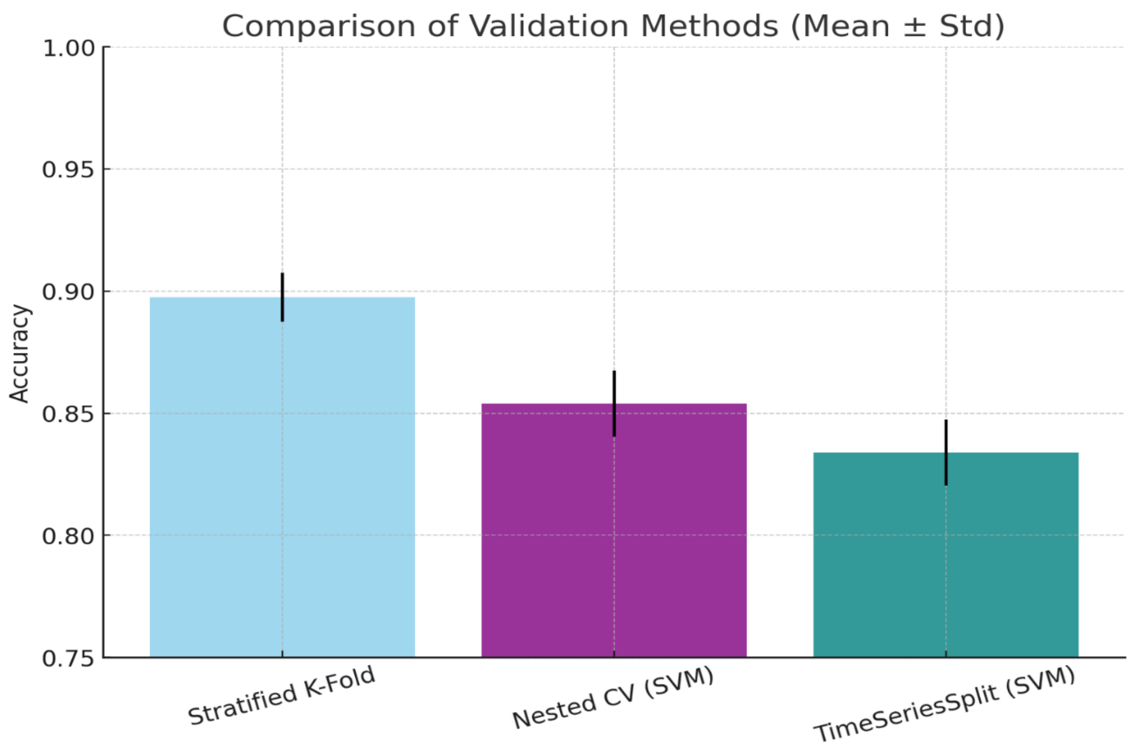

Fig 6: Comparison of Validation Methods (Mean ± Std)

Overall, the results confirm that the integration of CNN and Bi-LSTM yields a highly accurate and generalizable model for induction motor fault diagnosis. The model not only outperforms classical machine learning baselines but also demonstrates robustness across multiple validation strategies, making it suitable for real-world predictive maintenance applications.

IV. CONCLUSION

This study integrated and simulated a hybrid CNN-Bi-LSTM framework for induction motor fault diagnosis using multiple validation strategies. The model was evaluated against classical machine learning baselines (SVM, KNN, and MLP) and demonstrated superior accuracy, robustness, and generalization capability.

Under **5**-fold Stratified Cross-Validation, the CNN-Bi-LSTM achieved the highest mean accuracy of 93% ± 0.01, consistently outperforming the MLP (92% ± 0.01), KNN (89% ± 0.01), and SVM (85% ± 0.01). This highlights its strong ability to capture both spatial fault signatures and temporal dependencies in motor vibration signals.

When assessed with Nested Cross-Validation, which includes hyperparameter tuning, the framework maintained stable performance, confirming its generalizability and reducing the risk of overfitting to any particular data partition. Furthermore, evaluation with TimeSeriesSplit validation, which preserves temporal order, revealed that the model adapts effectively to sequential data streams, an important requirement for real-time predictive maintenance applications.

Taken together, these results establish that the proposed hybrid CNN-Bi-LSTM model not only surpasses conventional approaches in accuracy but also demonstrates consistent reliability across diverse validation protocols. This robustness indicates the framework's readiness for deployment in industrial environments, where dependable fault diagnosis is essential for minimizing downtime and optimizing maintenance costs.

REFERENCES

[1] A. Bellini, F. Filippetti, C. Tassoni, and G. A. Capolino, "Advances in diagnostic techniques for induction machines," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 12, pp. 4109–4126, 2008.

[2] P. Zhang, Y. Du, T. G. Habetler, and B. Lu, "A survey of condition monitoring and protection methods for medium-voltage induction motors," *IEEE Transactions on Industry Applications*, vol. 47, no. 1, pp. 34–46, 2011.

[3] S. Ahmed, M. Farajzadeh-Zanjani, and A. Abu-Siada, "Fault diagnosis of induction motors using machine learning," *Energies*, vol. 13, no. 13, pp. 1–16, 2020.

[4] H. Shao, H. Jiang, X. Li, and S. Wu, “Intelligent fault diagnosis of rolling bearings using an improved deep auto-encoder,” *Measurement Science and Technology*, vol. 29, no. 11, p. 115005, 2018.

[5] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] S. Malhotra, T. TV, A. Ramakrishnan, et al., “Multi-sensor prognostics using an unsupervised health index based on LSTM encoder–decoder,” IEEE International Conference on Big Data, pp. 1–10, 2016.

[8] R. Kohavi, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” *IJCAI*, pp. 1137–1145, 1995.

APPENDIX

```
In[1]:import numpy as np
    import pandas as pd
    import glob
In[4]:cur_path = "/kaggle/input/fault-induction-motor-dataset/imbalance/"


In[5]:normal_file_names = glob.glob("/kaggle/input/fault-induction-motor-
    dataset/normal/"+'/normal/*.csv')
    imnormal_file_names_6g =  glob.glob(cur_path+'/imbalance/6g/*.csv')
    imnormal_file_names_10g = glob.glob(cur_path+'/imbalance/10g/*.csv')
    imnormal_file_names_15g = glob.glob(cur_path+'/imbalance\\15g/*.csv')
    imnormal_file_names_20g = glob.glob(cur_path+'/imbalance\\20g/*.csv')
    imnormal_file_names_25g = glob.glob(cur_path+'/imbalance\\25g/*.csv')
    imnormal_file_names_30g = glob.glob(cur_path+'/imbalance\\30g/*.csv')


In[6]:def dataReader(path_names):
    data_n = pd.DataFrame()
    for i in path_names:
    low_data = pd.read_csv(i,header=None)
    data_n = pd.concat([data_n,low_data],ignore_index=True)
    return data_n


In[7]:data_n = dataReader(normal_file_names)
    data_6g = dataReader(imnormal_file_names_6g)
    data_10g = dataReader(imnormal_file_names_10g)
    data_15g = dataReader(imnormal_file_names_15g)
    data_20g = dataReader(imnormal_file_names_20g)
    data_25g = dataReader(imnormal_file_names_25g)
    data_30g = dataReader(imnormal_file_names_30g)
```

```
In[8]:data_n.info()
     <class 'pandas.core.frame.DataFrame'>
     RangeIndex: 12250000 entries, 0 to 12249999
     Data columns (total 8 columns):
 #   Column  Dtype
---  ------  -----
 0   0       float64
 1   1       float64
 2   2       float64
 3   3       float64
 4   4       float64
 5   5       float64
 6   6       float64
 7   7       float64
dtypes: float64(8)
memory usage: 747.7 MB
```

```
In[10]:data_n.head(2)
Out[10]:
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	-0.79198	-0.32398	-0.45381	0.23364	-0.40261	-0.020101	-0.10464	-0.099248
1	-0.95877	0.70368	-0.28989	0.25874	-0.36562	-0.018753	-0.00452	0.038236

```
In[11]:data_n.describe()

Out[11]:
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
count	1.225000e+07	1.225000e+07	1.225000e+07	1.225000e+07	1.225000e+07	1.225000e+07	1.225000e+07	1.225000e+07
mean	1.565263e-04	7.892067e-03	-4.660431e-04	3.732063e-04	1.483657e-02	3.021278e-03	1.877392e-02	1.227750e-02
std	1.711226e+00	8.536628e-01	4.174455e-01	1.837692e-01	6.248925e-01	3.479750e-02	4.133592e-01	1.755920e-01
min	-1.558800e+00	-4.483500e+00	-3.417000e+00	-2.171700e+00	-2.956600e+00	-3.733500e-01	-2.865800e+00	-3.369100e+00
25%	-7.135100e-01	-5.590000e-01	-2.866900e-01	-8.650100e-02	-3.302000e-01	-2.157700e-02	-2.295400e-01	-1.245400e-01
50%	-6.491500e-01	7.351000e-02	1.331900e-02	8.434800e-04	2.170600e-02	4.061300e-03	1.297300e-02	-6.496600e-03
75%	-5.384100e-01	6.521300e-01	2.979900e-01	8.529200e-02	3.465300e-01	2.810800e-02	2.617800e-01	1.332000e-01
max	5.107800e+00	2.367200e+00	4.099800e+00	1.933000e+00	3.508700e+00	2.812800e-01	2.836000e+00	8.529000e-01

```
In[12]:def downSampler(data,a,b):
    """
    data = data
    a = start index
    b = sampling rate
    """
    data_decreased = pd.DataFrame()
    x = b
    for i in range(int(len(data)/x)):
        data_decreased =    data_decreased.append(data.iloc[a:b,:].sum()/x,ignore_index=True)
        a += x
        b += x
    return data_decreased


In[13]:data_n = downSampler(data_n, 0, 5000)
       data_6g =  downSampler(data_6g, 0, 5000)
       data_10g = downSampler(data_10g, 0, 5000)
       data_15g = downSampler(data_15g, 0, 5000)
       data_20g = downSampler(data_20g, 0, 5000)
       data_25g = downSampler(data_25g, 0, 5000)
       data_30g = downSampler(data_30g, 0, 5000)
In[14]:data_n
```

Out[14]:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0.068100	0.011065	0.017430	0.001620	-0.059850	0.000868	-0.088720	0.010209
1	-0.045139	0.015286	-0.010404	-0.000644	0.426827	0.005168	0.155058	0.013550
2	-0.064635	0.029477	0.002314	0.001339	0.232491	0.005660	0.390845	0.009958
3	0.089400	-0.002910	0.002770	-0.002331	-0.116512	-0.003224	0.219854	0.012501
4	-0.070240	0.008164	-0.012449	0.002579	0.367824	0.008202	0.343822	0.010695
...	...	...	...	...	...	...	...	...
2445	0.014942	0.010274	0.000469	0.002580	0.138325	0.026142	0.336499	0.009176
2446	-0.078864	0.016197	0.001844	-0.000609	0.105457	0.016553	0.583317	0.011419
2447	-0.026635	0.000749	0.000612	0.002510	0.044352	0.010177	0.540934	0.012321
2448	0.016440	-0.018926	-0.010451	-0.003154	-0.053072	-0.011704	0.407358	0.012750
2449	0.026846	0.016300	-0.000227	-0.000158	-0.247938	-0.037531	-0.245142	0.011498

```
2450 rows × 8 columns
```

```
In[15]:y_1 = pd.DataFrame(np.ones(int(len(data_n)),dtype=int))
       y_2 = pd.DataFrame(np.zeros(int(len(data_6g)),dtype=int))
       y_3 = pd.DataFrame(np.full((int(len(data_10g)),1),0))
       y_4 = pd.DataFrame(np.full((int(len(data_15g)),1),0))
       y_5 = pd.DataFrame(np.full((int(len(data_20g)),1),0))
       y_6 = pd.DataFrame(np.full((int(len(data_25g)),1),0))
       y_7 = pd.DataFrame(np.full((int(len(data_30g)),1),0))
       y =  pd.concat([y_1,y_2,y_3,y_4,y_5,y_6,y_7], ignore_index=True)
       y
```

Out [15]:

	0
0	1
1	1
2	1
3	1
4	1
...	...
7295	0
7296	0
7297	0
7298	0
7299	0

7300 rows × 1 columns

```
In[17]:data =    pd.concat([data_n,data_6g,data_10g,data_15g,data_20g,data_25g,data_30g],ignor   e_index=True)

In[18]:data.head(5)
```

Out[18]:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0.068100	0.011065	0.017430	0.001620	-0.059850	0.000868	-0.088720	0.010209
1	-0.045139	0.015286	-0.010404	-0.000644	0.426827	0.005168	0.155058	0.013550
2	-0.064635	0.029477	0.002314	0.001339	0.232491	0.005660	0.390845	0.009958
3	0.089400	-0.002910	0.002770	-0.002331	-0.116512	-0.003224	0.219854	0.012501
4	-0.070240	0.008164	-0.012449	0.002579	0.367824	0.008202	0.343822	0.010695

```
In[19]:from sklearn.model_selection import train_test_split
       X_train, X_test, y_train, y_test = train_test_split(data, y,
     test_size=0.25, shuffle=True)
```

```
In[20]:print("Shape of Train Data : {}".format(X_train.shape))
       print("Shape of Test Data : {}".format(X_test.shape))
       Shape of Train Data : (5475, 8)
       Shape of Test Data : (1825, 8)

In[24]:print("Shape of Train Data : {}".format(y_train.shape))
       print("Shape of Test Data : {}".format(y_test.shape))
       Shape of Train Data : (5475, 1)
       Shape of Test Data : (1825, 1)

In[46]:unique_values, value_counts = np.unique(y_train, return_counts=True)
       for value, count in zip(unique_values, value_counts):
       print(f"Value: {value}, Count: {count}")
       Value: 0, Count: 3635
       Value: 1, Count: 1840

In[49]:from imblearn.over_sampling import SMOTE

In[51]:# Apply resampling technique (e.g., SMOTE)
       smote = SMOTE()
       X_resampled, y_resampled = smote.fit_resample(data,y)

In[52]:# Split the data into training and testing sets
       X_train_s, X_test_s, y_train_s, y_test_s =

  train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

In[53]:print("Shape of Train Data : {}".format(X_train_s.shape))
       print("Shape of Test Data : {}".format(X_test_s.shape))
       Shape of Train Data : (7760, 8)
       Shape of Test Data : (1940, 8)

In[54]:unique_values, value_counts = np.unique(y_train_s, return_counts=True)
       for value, count in zip(unique_values, value_counts):
       print(f"Value: {value}, Count: {count}")
       Value: 0, Count: 3876
       Value: 1, Count: 3884
```

```
In[21]:from sklearn.svm import SVC
svm = SVC(random_state = 1)
svm.fit(X_train,y_train)
print("SVM accuracy is {} on Train Dataset".format(svm.score(X_train,y_train)))
print("SVM accuracy is {} on Test Dataset".format(svm.score(X_test,y_test)))
/opt/conda/lib/python3.7/site-packages/sklearn/utils/validation.py:63: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  return f(*args, **kwargs)
SVM accuracy is 0.8615525114155251 on Train Dataset
SVM accuracy is 0.8531506849315068 on Test Dataset
```

```
In[22]:from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 3) #n_neighbors = k
knn.fit(X_train,y_train)
print("k={}NN Accuracy on Train Data: {}".format(3,knn.score(X_train,y_train)))
print("k={}NN Accuracy on Test Data: {}".format(3,knn.score(X_test,y_test)))
/opt/conda/lib/python3.7/site-packages/sklearn/neighbors/_classification.py:179: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  return self._fit(X, y)
k=3NN Accuracy on Train Data: 0.930228310502283
k=3NN Accuracy on Test Data: 0.8854794520547945
```

```
In[23]:from sklearn.neural_network import MLPClassifier
nn = MLPClassifier(solver='lbfgs')
nn.fit(X_train,y_train)
print("MLP Accuracy on Train Data: {}".format(nn.score(X_train,y_train)))
print("MLP Accuracy on Test Data: {}".format(nn.score(X_test,y_test)))
/opt/conda/lib/python3.7/site-packages/sklearn/utils/validation.py:63: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  return f(*args, **kwargs)
MLP Accuracy on Train Data: 0.9347945205479452
MLP Accuracy on Test Data: 0.9238356164383562
/opt/conda/lib/python3.7/site-packages/sklearn/neural_network/_multilayer_perceptron.py:500: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

```
In[25]:import numpy as np
       import tensorflow as tf
       from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Bidirectional, LSTM, Dense, Flatten

In[55]:# Convert DataFrames to NumPy arrays
       X_train = X_train_s.values
       y_train = y_train_s.values


       # Reshape the input data to match the Conv1D input shape
       X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)

In[56]:# Evaluate the model on the test set
       X_test = X_test_s.values
       y_test = y_test_s.values

       X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

In[57]:# Define the model
model = Sequential() model.add(Conv1D(filters=32, kernel_size=3, activation='relu',   input_shape=(X_train.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2))
model.add(Bidirectional(LSTM(64, return_sequences=True)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Assuming binary classification

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
Epoch 1/100
194/194 [==============================] - 4s 8ms/step - loss: 0.5608 - accuracy: 0.6907 - val_loss: 0.3318 -
val_accuracy: 0.8499
Epoch 2/100
194/194 [==============================] - 1s 5ms/step - loss: 0.3048 - accuracy: 0.8658 - val_loss: 0.3016 -
val_accuracy: 0.8666
Epoch 3/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.2868 - accuracy: 0.8764 - val_loss: 0.2850 -
val_accuracy: 0.8769

Epoch 4/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2896 - accuracy: 0.8774 - val_loss: 0.3274 -
val_accuracy: 0.8570

Epoch 5/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2679 - accuracy: 0.8890 - val_loss: 0.2674 -
val_accuracy: 0.8795

Epoch 6/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2659 - accuracy: 0.8816 - val_loss: 0.2816 -
val_accuracy: 0.8731

Epoch 7/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2435 - accuracy: 0.8890 - val_loss: 0.2533 -
val_accuracy: 0.8840

Epoch 8/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2395 - accuracy: 0.8949 - val_loss: 0.2623 -
val_accuracy: 0.8821

Epoch 9/100

194/194 [==============================] - 1s 6ms/step - loss: 0.2308 - accuracy: 0.8991 - val_loss: 0.2444 -
val_accuracy: 0.8853

Epoch 10/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2382 - accuracy: 0.8908 - val_loss: 0.2327 -
val_accuracy: 0.8956

Epoch 11/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2209 - accuracy: 0.9040 - val_loss: 0.2823 -
val_accuracy: 0.8666

Epoch 12/100

194/194 [==============================] - 1s 6ms/step - loss: 0.2303 - accuracy: 0.8984 - val_loss: 0.2208 -
val_accuracy: 0.9008

Epoch 13/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2319 - accuracy: 0.8990 - val_loss: 0.2215 -
val_accuracy: 0.8963

Epoch 14/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2071 - accuracy: 0.9094 - val_loss: 0.2155 -
val_accuracy: 0.9008

Epoch 15/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2094 - accuracy: 0.9017 - val_loss: 0.2256 -
val_accuracy: 0.9034

Epoch 16/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.2165 - accuracy: 0.9074 - val_loss: 0.2084 -
val_accuracy: 0.9014

Epoch 17/100

194/194 [==============================] - 1s 5ms/step - loss: 0.2091 - accuracy: 0.9073 - val_loss: 0.2187 -
val_accuracy: 0.9001

Epoch 18/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1983 - accuracy: 0.9116 - val_loss: 0.2000 -
val_accuracy: 0.9072

Epoch 19/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1971 - accuracy: 0.9145 - val_loss: 0.2182 -
val_accuracy: 0.9053

Epoch 20/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1965 - accuracy: 0.9150 - val_loss: 0.1891 -
val_accuracy: 0.9156

Epoch 21/100

194/194 [==============================] - 1s 6ms/step - loss: 0.1863 - accuracy: 0.9184 - val_loss: 0.2034 -
val_accuracy: 0.9059

Epoch 22/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1810 - accuracy: 0.9270 - val_loss: 0.2034 -
val_accuracy: 0.8976

Epoch 23/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1900 - accuracy: 0.9194 - val_loss: 0.2131 -
val_accuracy: 0.9156

Epoch 24/100

194/194 [==============================] - 1s 6ms/step - loss: 0.1852 - accuracy: 0.9220 - val_loss: 0.1859 -
val_accuracy: 0.9111

Epoch 25/100

194/194 [==============================] - 1s 6ms/step - loss: 0.1790 - accuracy: 0.9158 - val_loss: 0.1814 -
val_accuracy: 0.9169

Epoch 26/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1790 - accuracy: 0.9198 - val_loss: 0.1752 -
val_accuracy: 0.9227

Epoch 27/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1564 - accuracy: 0.9299 - val_loss: 0.1741 -
val_accuracy: 0.9175

Epoch 28/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1677 - accuracy: 0.9275 - val_loss: 0.1788 -
val_accuracy: 0.9182

Epoch 29/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1646 - accuracy: 0.9275 - val_loss: 0.1691 -
val_accuracy: 0.9285

Epoch 30/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.1655 - accuracy: 0.9271 - val_loss: 0.1656 -
val_accuracy: 0.9246

Epoch 31/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1654 - accuracy: 0.9311 - val_loss: 0.1916 -
val_accuracy: 0.9124

Epoch 32/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1541 - accuracy: 0.9313 - val_loss: 0.1590 -
val_accuracy: 0.9259

Epoch 33/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1631 - accuracy: 0.9271 - val_loss: 0.1561 -
val_accuracy: 0.9233

Epoch 34/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1586 - accuracy: 0.9230 - val_loss: 0.1621 -
val_accuracy: 0.9201

Epoch 35/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1559 - accuracy: 0.9315 - val_loss: 0.1686 -
val_accuracy: 0.9246

Epoch 36/100

194/194 [==============================] - 1s 6ms/step - loss: 0.1491 - accuracy: 0.9346 - val_loss: 0.1502 -
val_accuracy: 0.9343

Epoch 37/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1601 - accuracy: 0.9350 - val_loss: 0.1660 -
val_accuracy: 0.9272

Epoch 38/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1396 - accuracy: 0.9381 - val_loss: 0.1599 -
val_accuracy: 0.9253

Epoch 39/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1489 - accuracy: 0.9360 - val_loss: 0.1644 -
val_accuracy: 0.9195

Epoch 40/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1490 - accuracy: 0.9318 - val_loss: 0.1403 -
val_accuracy: 0.9375

Epoch 41/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1438 - accuracy: 0.9339 - val_loss: 0.1504 -
val_accuracy: 0.9259

Epoch 42/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1539 - accuracy: 0.9318 - val_loss: 0.1667 -
val_accuracy: 0.9207

Epoch 43/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1510 - accuracy: 0.9349 - val_loss: 0.1465 -
val_accuracy: 0.9323

Epoch 44/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.1519 - accuracy: 0.9292 - val_loss: 0.1446 -
val_accuracy: 0.9336

Epoch 45/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1420 - accuracy: 0.9340 - val_loss: 0.1525 -
val_accuracy: 0.9259

Epoch 46/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1469 - accuracy: 0.9320 - val_loss: 0.1649 -
val_accuracy: 0.9298

Epoch 47/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1475 - accuracy: 0.9340 - val_loss: 0.1568 -
val_accuracy: 0.9278

Epoch 48/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1396 - accuracy: 0.9361 - val_loss: 0.1462 -
val_accuracy: 0.9336

Epoch 49/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1390 - accuracy: 0.9364 - val_loss: 0.1437 -
val_accuracy: 0.9362

Epoch 50/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1348 - accuracy: 0.9386 - val_loss: 0.1427 -
val_accuracy: 0.9304

Epoch 51/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1364 - accuracy: 0.9428 - val_loss: 0.1404 -
val_accuracy: 0.9369

Epoch 52/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1419 - accuracy: 0.9410 - val_loss: 0.1803 -
val_accuracy: 0.9156

Epoch 53/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1438 - accuracy: 0.9360 - val_loss: 0.1770 -
val_accuracy: 0.9272

Epoch 54/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1465 - accuracy: 0.9312 - val_loss: 0.1628 -
val_accuracy: 0.9253

Epoch 55/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1460 - accuracy: 0.9360 - val_loss: 0.1457 -
val_accuracy: 0.9291

Epoch 56/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1427 - accuracy: 0.9421 - val_loss: 0.1398 -
val_accuracy: 0.9356

Epoch 57/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1363 - accuracy: 0.9404 - val_loss: 0.1399 -
val_accuracy: 0.9394

Epoch 58/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.1309 - accuracy: 0.9435 - val_loss: 0.1590 -
val_accuracy: 0.9304

Epoch 59/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1348 - accuracy: 0.9391 - val_loss: 0.1409 -
val_accuracy: 0.9369

Epoch 60/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1474 - accuracy: 0.9314 - val_loss: 0.1386 -
val_accuracy: 0.9362

Epoch 61/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1375 - accuracy: 0.9371 - val_loss: 0.1519 -
val_accuracy: 0.9298

Epoch 62/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1355 - accuracy: 0.9417 - val_loss: 0.1441 -
val_accuracy: 0.9375

Epoch 63/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1497 - accuracy: 0.9308 - val_loss: 0.1370 -
val_accuracy: 0.9336

Epoch 64/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1293 - accuracy: 0.9457 - val_loss: 0.1432 -
val_accuracy: 0.9369

Epoch 65/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1332 - accuracy: 0.9383 - val_loss: 0.1656 -
val_accuracy: 0.9259

Epoch 66/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1545 - accuracy: 0.9310 - val_loss: 0.1408 -
val_accuracy: 0.9381

Epoch 67/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1441 - accuracy: 0.9418 - val_loss: 0.1294 -
val_accuracy: 0.9420

Epoch 68/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1435 - accuracy: 0.9353 - val_loss: 0.1874 -
val_accuracy: 0.9240

Epoch 69/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1388 - accuracy: 0.9414 - val_loss: 0.1365 -
val_accuracy: 0.9446

Epoch 70/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1320 - accuracy: 0.9447 - val_loss: 0.1432 -
val_accuracy: 0.9304

Epoch 71/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1296 - accuracy: 0.9432 - val_loss: 0.1339 -
val_accuracy: 0.9420

Epoch 72/100
```

```
194/194 [==============================] - 1s 6ms/step - loss: 0.1363 - accuracy: 0.9386 - val_loss: 0.1656 -
val_accuracy: 0.9201

Epoch 73/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1406 - accuracy: 0.9372 - val_loss: 0.1381 -
val_accuracy: 0.9362

Epoch 74/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1285 - accuracy: 0.9439 - val_loss: 0.1454 -
val_accuracy: 0.9349

Epoch 75/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1348 - accuracy: 0.9408 - val_loss: 0.1305 -
val_accuracy: 0.9427

Epoch 76/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1279 - accuracy: 0.9476 - val_loss: 0.1293 -
val_accuracy: 0.9388

Epoch 77/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1304 - accuracy: 0.9416 - val_loss: 0.1415 -
val_accuracy: 0.9349

Epoch 78/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1379 - accuracy: 0.9444 - val_loss: 0.1498 -
val_accuracy: 0.9291

Epoch 79/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1344 - accuracy: 0.9380 - val_loss: 0.1445 -
val_accuracy: 0.9369

Epoch 80/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1413 - accuracy: 0.9384 - val_loss: 0.1241 -
val_accuracy: 0.9485

Epoch 81/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1244 - accuracy: 0.9488 - val_loss: 0.1396 -
val_accuracy: 0.9317

Epoch 82/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1393 - accuracy: 0.9349 - val_loss: 0.1295 -
val_accuracy: 0.9433

Epoch 83/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1387 - accuracy: 0.9382 - val_loss: 0.1364 -
val_accuracy: 0.9388

Epoch 84/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1349 - accuracy: 0.9400 - val_loss: 0.1234 -
val_accuracy: 0.9446

Epoch 85/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1245 - accuracy: 0.9436 - val_loss: 0.1341 -
val_accuracy: 0.9401

Epoch 86/100
```

```
194/194 [==============================] - 1s 6ms/step - loss: 0.1324 - accuracy: 0.9428 - val_loss: 0.1291 -
val_accuracy: 0.9414

Epoch 87/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1229 - accuracy: 0.9468 - val_loss: 0.1278 -
val_accuracy: 0.9394

Epoch 88/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1251 - accuracy: 0.9482 - val_loss: 0.1341 -
val_accuracy: 0.9336

Epoch 89/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1292 - accuracy: 0.9448 - val_loss: 0.1329 -
val_accuracy: 0.9375

Epoch 90/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1260 - accuracy: 0.9468 - val_loss: 0.1237 -
val_accuracy: 0.9439

Epoch 91/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1241 - accuracy: 0.9439 - val_loss: 0.1465 -
val_accuracy: 0.9311

Epoch 92/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1343 - accuracy: 0.9425 - val_loss: 0.1239 -
val_accuracy: 0.9446

Epoch 93/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1225 - accuracy: 0.9467 - val_loss: 0.1300 -
val_accuracy: 0.9420

Epoch 94/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1357 - accuracy: 0.9415 - val_loss: 0.1210 -
val_accuracy: 0.9459

Epoch 95/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1294 - accuracy: 0.9450 - val_loss: 0.1258 -
val_accuracy: 0.9407

Epoch 96/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1206 - accuracy: 0.9459 - val_loss: 0.1287 -
val_accuracy: 0.9427

Epoch 97/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1342 - accuracy: 0.9400 - val_loss: 0.1214 -
val_accuracy: 0.9472

Epoch 98/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1275 - accuracy: 0.9422 - val_loss: 0.1241 -
val_accuracy: 0.9452

Epoch 99/100

194/194 [==============================] - 1s 5ms/step - loss: 0.1192 - accuracy: 0.9463 - val_loss: 0.1288 -
val_accuracy: 0.9394

Epoch 100/100
```