Team - DA666

Table of Contents

Abstract

This report shows the development of various Machine learning models to predict the state of an automated manufacturing unit based on data from 25 sensors. To achieve this goal, various basic machine learning and deep learning models were explored, followed by time series analysis and sequencing models. As opposed to these, a convolutional-based approach was implemented, which surpassed all other models and gave optimistic results.

Introduction

Automated manufacturing units have become increasingly popular due to their ability to improve efficiency and reduce costs. These units rely heavily on sensors to monitor various aspects of the production process, such as temperature, pressure, and vibration. The data collected from these sensors can provide valuable insights into the performance of the unit, which can be used to optimize production and prevent costly downtime.

However, with the large amount of data generated by these sensors, making sense of the information and identifying potential issues can be challenging. This is where data analytics comes in. By analyzing the data collected from sensors, we can identify patterns and anomalies that may indicate a problem with the unit. This can allow us to take proactive steps to prevent downtime and ensure that the unit continues to operate at peak performance.

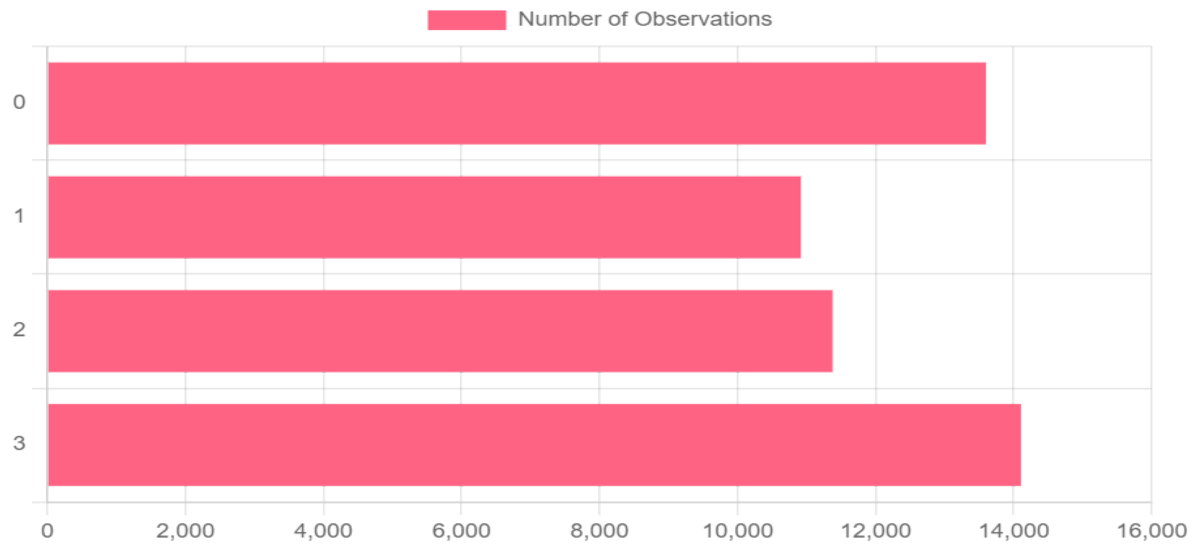
In this study, we focus on developing a model that can predict the state of an automated manufacturing unit using data from 25 sensors. The state of the unit is classified as 'no risk' 'low risk', 'medium risk', or 'catastrophic' and is recorded every 10 minutes. By accurately predicting the state of the unit, we can take proactive steps to prevent issues and enhance the performance of the system, nonetheless making an AI-based model which predicts states and removes human intervention on a large scale.

About the Data

The size of the provided dataset has a total of 50000 samples, with each having the data of 25 sensors for a time span of 10 minutes and the state of each sample, respectively.
We have segregated the output labels into the following four states:

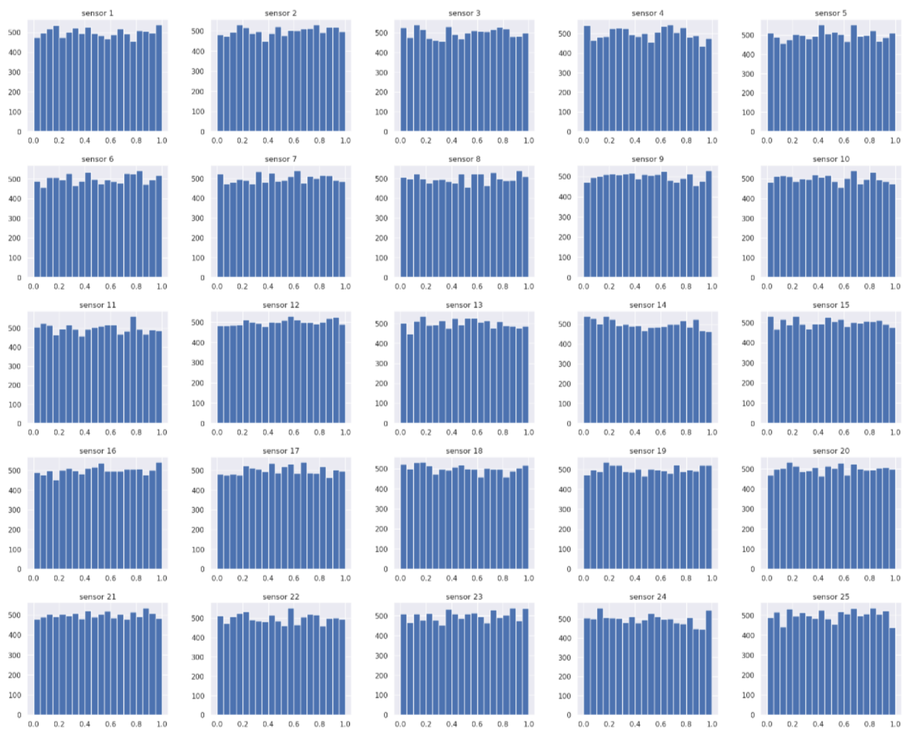
Labels	*Count*
0 ~' No Risk'	13599
1~ 'Low Risk'	10917
2~ 'Medium Risk'	11377

3~ ’Catastrophic’	14107



From the above distribution of output states, we can say that the data is balanced.

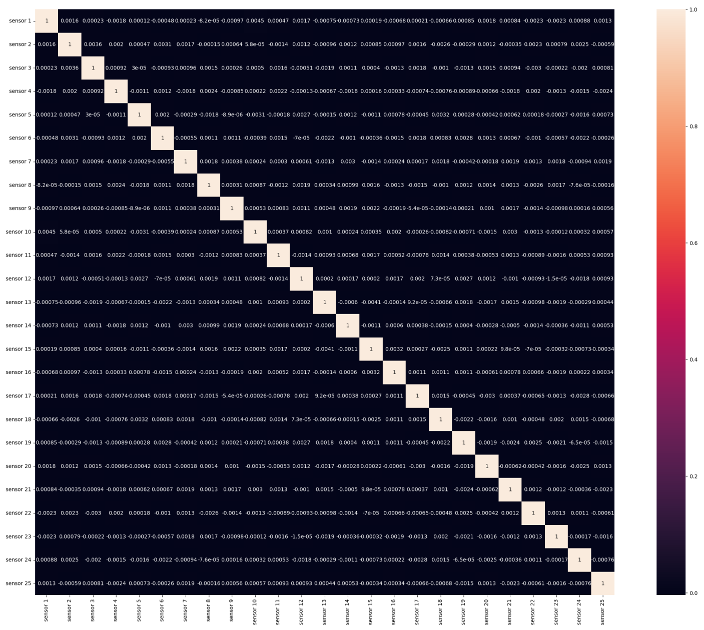
The distribution of sensor values for a given class category is shown as an histogram, where each plot represents a sensor (there are 25 such histograms) and each histogram has the value 0 to 1 in the x-axis and the y-axis demonstrates the frequency of occurrence of the values belonging to those bins.



From the above graph, we can conclude the individual sensor data is uniformly distributed. This is also observed for the other 3 class categories and thus shows that on a scale of the individual sensor values, they cannot be distinguished by some well-defined boundary.

As evident from the above subplots, the sensor-wise data spanning over 300 random chunks is uniform.

Correlation Heatmap:



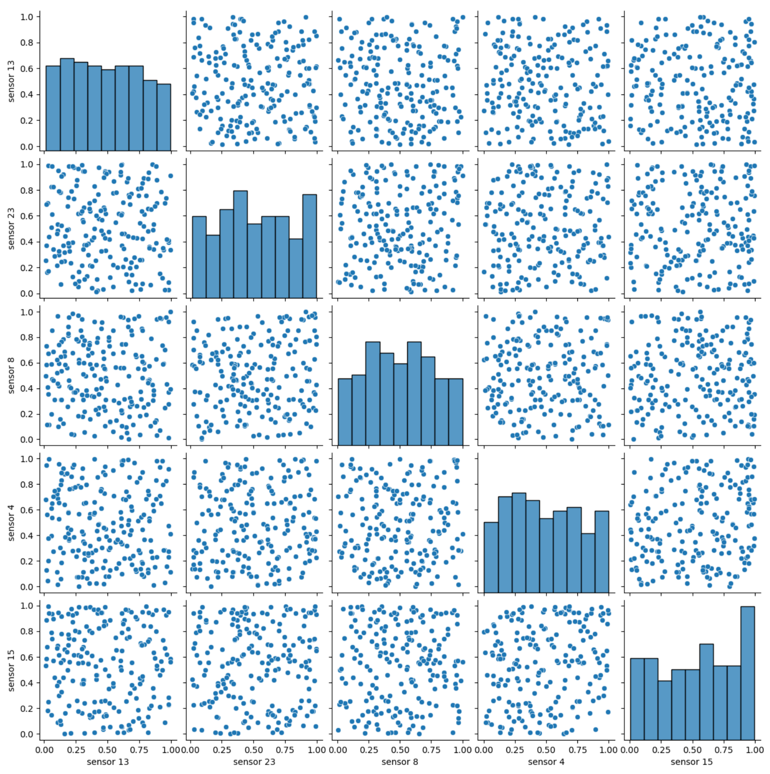
We have also calculated the **standard deviation** and **mean** of each sensor which are **0.29** and **0.5**, respectively. This shows that all the sensor data are distributed uniformly, and surprisingly, all 25 sensors give the same value of mean and standard deviation.

Now, plotting the **pair plot** for random five sensors to find any possible correlation or dependency among sensors in the dataset.

Pairplot for random five sensors:

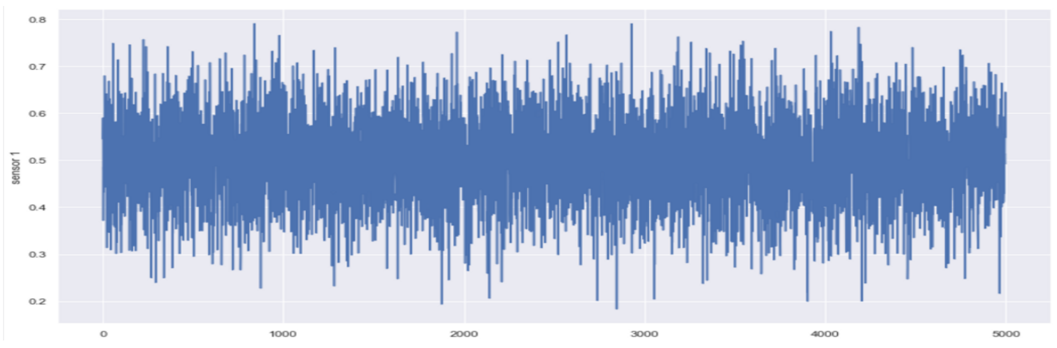
From the below data analysis of randomly chosen five sensors, we can safely conclude the following:

- The provided data is not skewed and is mostly uniformly distributed.
- The data is not related pairwise to any feature.
- There are no trends or seasonality in the sensor-wise data.



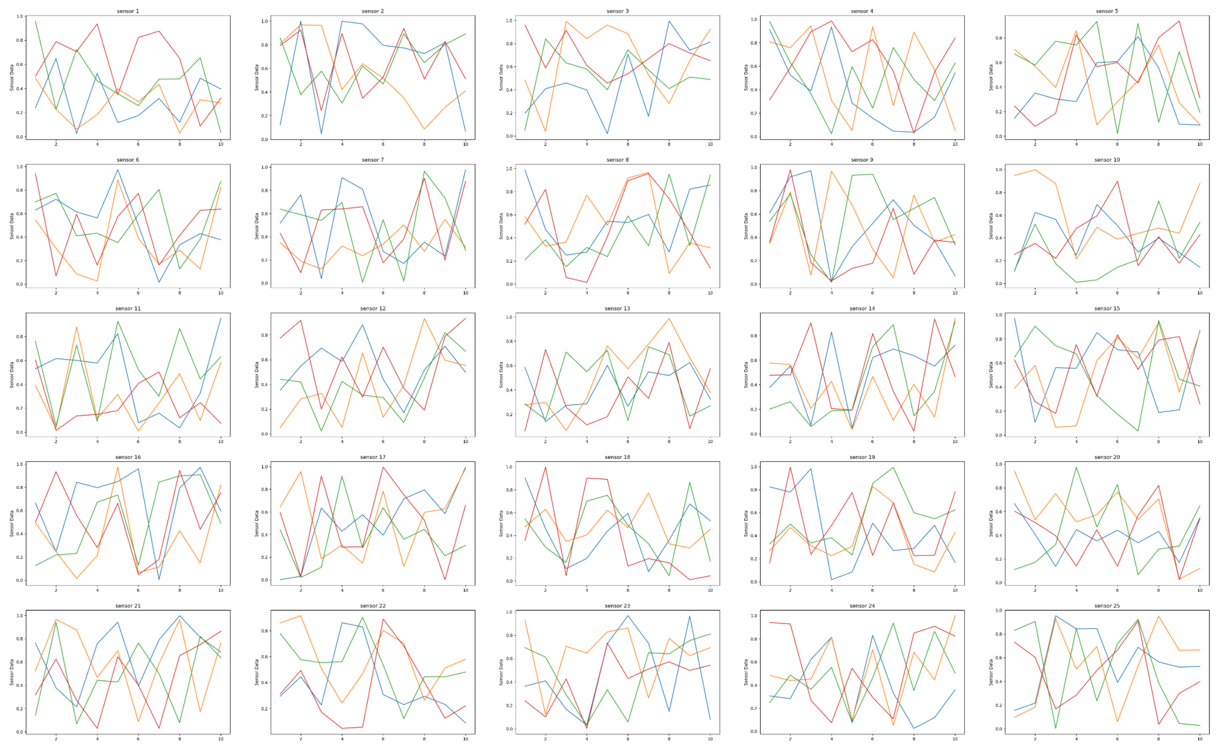
The reason for building the above plots was to see if there were any cluster formations that could distinguish the classes. But so far we found that the data is spread very uniformly over the data space. Hence traditional ML approaches would not be able to find any primitive boundary of classification, hence our first step is to implement a simple DNN (Deep Neural Network), to see if this model or its intermediary layers could capture any transformation of the dataset that could help in an easy segregation. But first, we wanted to check for any time series trends in the data which could impact the machine state.

We plot the trend of a randomly chosen sensor, “Sensor 1” in this case, over 5000 random chunks.



Performing the Augmented Dickey-Fuller Test, the obtained p-value is less than 0.05. This result disproves the existence of any trends, seasonality, or cyclicity in the series. Thus we assume that each machine state can be treated to be independent and only depends on the 10 minute chunk and not on the previous occurrence of the state type.

Plotting the Sensor Reading v/s Time for the label ‘3’ (picking 4 eg. at random):



A similar noisy graph was deduced for each of the observations. We now move on to the development of our models.

Model Development

Basic Machine learning Models

The required model has to classify the manufacturing unit state i.e., a multi-classification model is to be developed, and the model should also learn the correlation between the sequential data in each sample.

We started exploring different models by building models on Random Forest Classifier, SVMs and XGBoost, which are the most widely used algorithms for multi-class classification.

Random Forest Classifier, Support Vector Machines, XGBoost

To process the sensor data collected every minute in 10-minute intervals, we took two approaches.

First we took the average of the 10 min data to form a single 25 features tensor corresponding to a single output label. Next we concatenated the data from each minute, resulting in a dataset with 250 features. This means that the data is in the following format : [..25 features (min 1)…, ….25 features (min 2…, and so on till (min 10)]. We then transformed the prediction label into a **one-hot encoding** format, where the label for "no risk" was represented as {1,0,0,0}, "low risk" as {0,1,0,0}, "medium risk" as {0,0,1,0}, and "catastrophic" as {0,0,0,1}.

We trained the dataset obtained after concatenating the sensor data with basic machine learning models such as Random Forest, Support Vector Machine, and XG Boost. However, the performance of these models was not satisfactory, with the best accuracy obtained being only around 0.30.

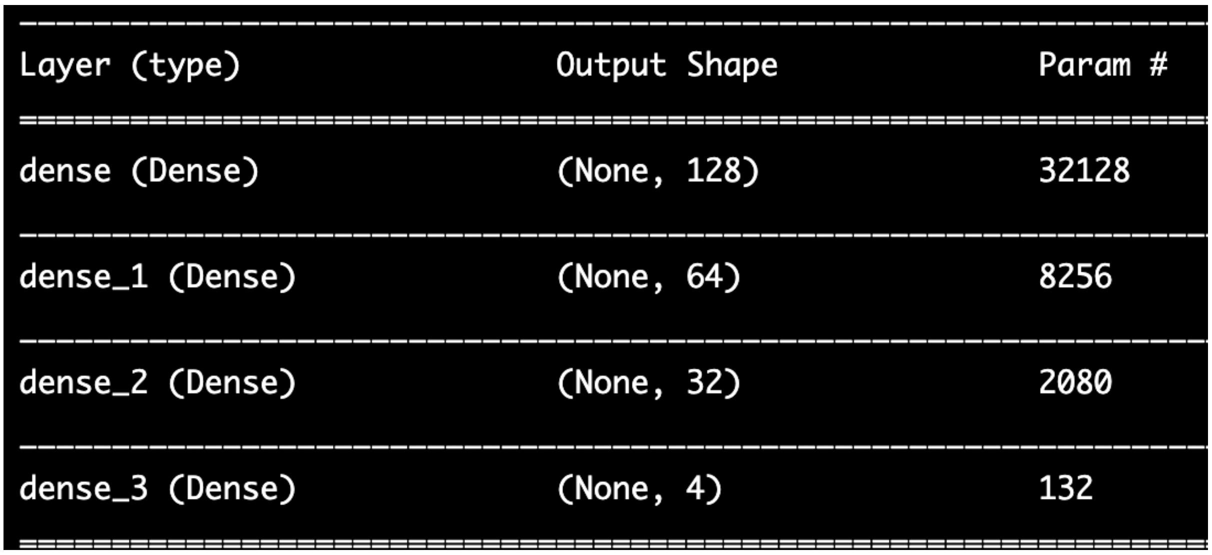
Model Name	Accuracy (in percentage)
Random Forest	27.3
Support Vector machine	28.1
XG Boost	30.1

This suggested that the task of predicting the state of the manufacturing unit based on the sensor data is challenging and requires more sophisticated machine-learning approaches. It may be that the current models are not able to capture the complex relationships between the sensor data and the state of the unit or that there are limitations in the quality or quantity of the available data.

Deep Neural Networks

In an attempt to improve the accuracy of our predictions for the manufacturing unit state, we explored the use of deep neural networks. Specifically, we trained the dataset using different numbers of hidden layers of neural networks with various size combinations.

DNN Model Architecture



```
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
dense (Dense)                (None, 128)               32128
________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
________________________________________________________________
dense_3 (Dense)              (None, 4)                 132
================================================================
```

Despite the use of deep neural networks, the accuracy score did not show a significant improvement and remained around 0.3. Moreover, the neural networks often returned only one label for the entire dataset, indicating that the models were not able to effectively capture the variability and complexity of the sensor data over time. We found that the DNN model was not effectively learning from the dataset. This was indicated by the fact that the loss metric did not decrease significantly over the course of the training epochs, and the validation accuracy did not improve either.

There could be several reasons for this, such as the complexity of the model architecture, the size, and quality of the dataset, or the choice of hyperparameters. It is possible it was not able to effectively capture the complex patterns in the data, or make any non-linear transformations that could create a well-defined segregation boundary.

Despite our efforts to fine-tune the DNN model by adjusting the number of layers, the size of the layers, and the learning rate, we were not able to achieve a significant improvement in performance.
Since we were considering the average over the 10 min interval as input to our models in some cases, it must have been missing out on the time dependency by applying equal weightage to each minute. What should have been intuitively possible is that the 10th min sensor values should have a higher weightage as compared to the 1st min sensor values. To capture a better representation of this, we pass the 10 min data of 25 features each to an RNN-based model like GRU, Vanilla RNN, and LSTM.
This would help in generating an output tensor which would be a better 1D representation of the data as opposed to a mean value calculation.

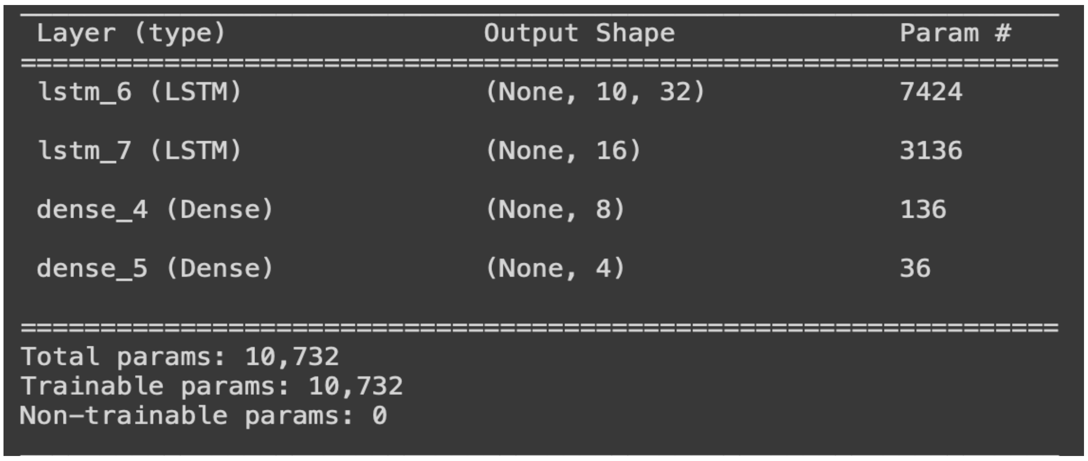
Recurrent Neural Networks

In an attempt to increase the accuracy of the predictions of the unit's state, we explored recurrent neural networks.

As each sample has 10-time steps, RNN Networks are not designed to learn the Long term dependencies in the data. So to process the sensor data collected every minute in 10-minute intervals, the sample data is transformed into a tensor of size [batch size, time steps, sensors]=[50000,10,25], which will be given as input into recurrent neural network layers. We transformed the prediction label into a one-hot encoding format, where the label for "no risk" was represented as {1,0,0,0}, "low risk" as {0,1,0,0}, "medium risk" as {0,0,1,0}, and "catastrophic" as {0,0,0,1}.

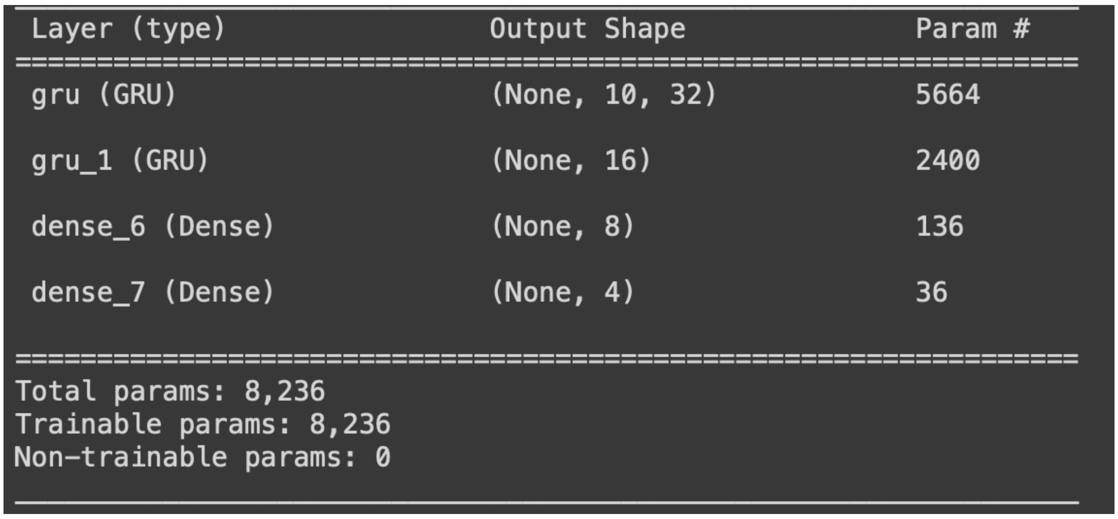
As each sample has 10-time steps, RNN Networks are not designed to learn the Long term dependencies in the data. So we have developed models using suitable layers, i.e., Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

Long Short-Term Memory Architecture



```
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_6 (LSTM)               (None, 10, 32)            7424

 lstm_7 (LSTM)               (None, 16)                3136

 dense_4 (Dense)             (None, 8)                 136

 dense_5 (Dense)             (None, 4)                 36

=================================================================
Total params: 10,732
Trainable params: 10,732
Non-trainable params: 0
```

Gated Recurrent Unit Architecture

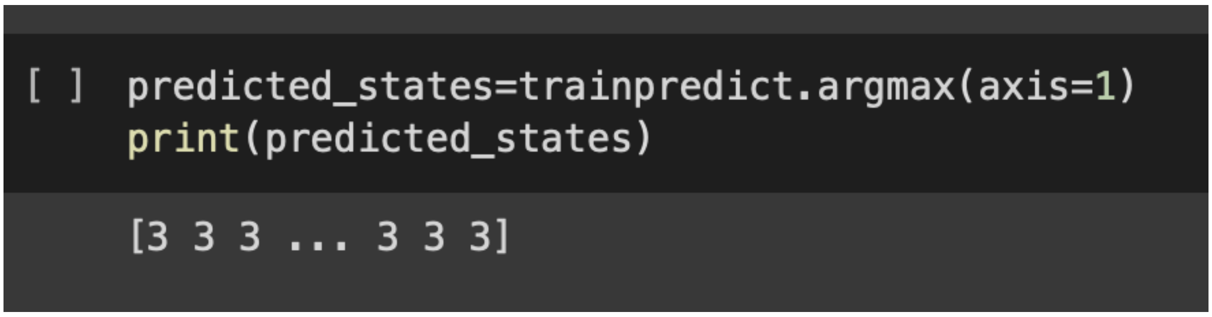


```
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 10, 32)            5664

 gru_1 (GRU)                 (None, 16)                2400

 dense_6 (Dense)             (None, 8)                 136

 dense_7 (Dense)             (None, 4)                 36

=================================================================
Total params: 8,236
Trainable params: 8,236
Non-trainable params: 0
```

Model Name	Accuracy (in percentage)

LSTM	28.33
GRU	28.36

Both the RNNs have failed to learn the relation between the time steps of the data as both models have an accuracy of nearly 28.33 percent and both the models are only predicting a single state of the manufacturing unit which indicates that recurrent networks are unable to learn the interrelation within time steps in sample data. This was similar to the case where we applied a simple DNN for training.



```
[ ] predicted_states=trainpredict.argmax(axis=1)
    print(predicted_states)

    [3 3 3 ... 3 3 3]
```

So we have proceeded further with experimenting with convolutional architectures capable of sliding over the features and having a shared-weight architecture and may learn the interrelation in time steps of sample data.

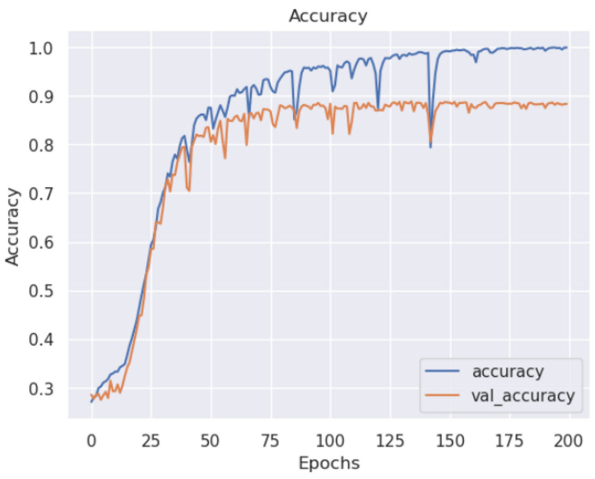
1-D Convolution: Combinations of Conv1D and Dense Layers

With the intuition to get the pattern from some unobvious relationships among the rows of the sensors data, we concatenated each row side by side for each of the chunks of 10 observations, side by side. The labels were one-hot encoded, and the input size to the model was (1, 250). Some of the explicitly defined parameters are as follows:

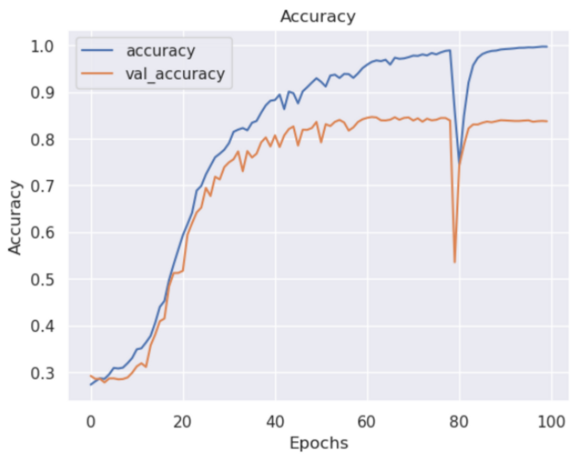
Loss Function	Categorical Cross Entropy
Optimizer	Adam (learning rate = 1e-3)
Epochs	100
Batch Size	2048
Validation Split	0.25
Output layer Activation Function	Softmax

Various models were tried using the above fitting parameters, some of them including but not limited to:

- Single Conv1D layer with 32 filters
- Single Conv1D layer with a high number of filters, such as 128
- Multiple Conv1D layers with 32, 64, 128 filters with varying stride lengths



Out of all these models, Single Conv1D layer with a relatively high number of filters performed the best in terms of validation accuracy, with 128 filters producing an accuracy of 88.6%.



However, the model suffers from rapidly fluctuating validation loss problems. A relatively stable model can be created using 64 filters.

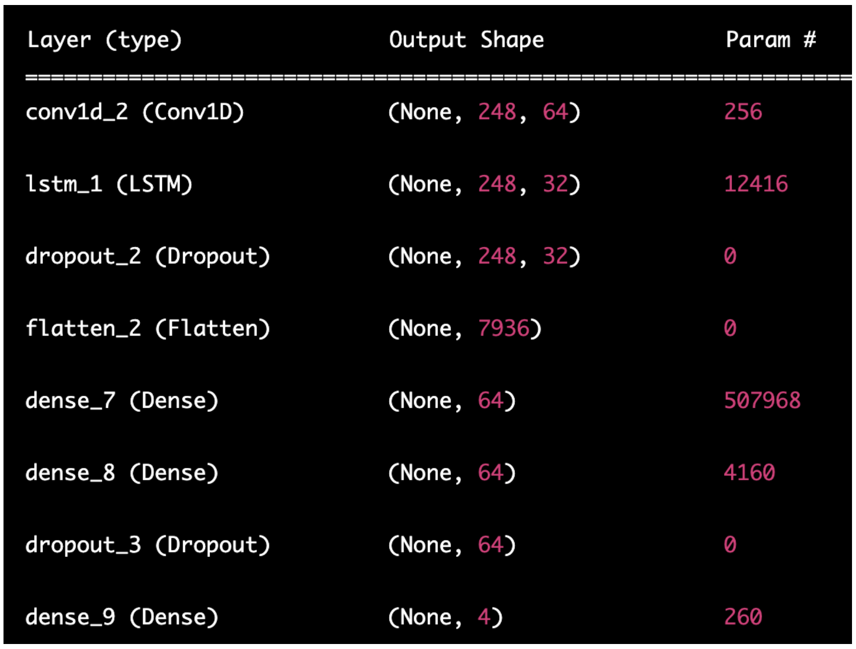
Applying large stride lengths to capture sensor-wide and time-dependent trends, as well as applying multiple Conv1D layers, proves to be counter-intuitive, with the model rapidly learning and overfitting the data, resulting in a poor overall validation score performance. In our example, we implemented 2 Conv1D layers with 64 and 32 filters with 1 and 2 strides, respectively.

This model does not pose any significant improvement and motivates us to explore higher dimensional convolutional models.

Convolution 1D + LSTM Model

After our initial attempts with basic machine learning models and various types of neural networks, we decided to try a new approach by adding a convolutional layer (Conv 1D) before the LSTM layer. This modification enabled the network to learn hierarchical representations of the input data, which can be particularly useful for time-series data.

Conv 1D + LSTM model architecture:


```
Layer (type)                Output Shape              Param #
=================================================================
conv1d_2 (Conv1D)           (None, 248, 64)           256

lstm_1 (LSTM)               (None, 248, 32)           12416

dropout_2 (Dropout)         (None, 248, 32)           0

flatten_2 (Flatten)         (None, 7936)              0

dense_7 (Dense)             (None, 64)                507968

dense_8 (Dense)             (None, 64)                4160

dropout_3 (Dropout)         (None, 64)                0

dense_9 (Dense)             (None, 4)                 260
```
:

Using this modified architecture, we were able to achieve an accuracy score of ~55%, which represents a significant improvement over our previous results (where no convolution was used). This improvement in accuracy suggests that the Conv1D layer was able to effectively extract relevant features from the sensor data, which the subsequent LSTM layer was able to use to make accurate predictions about the state of the manufacturing unit.

Overall, this approach shows promise for improving the accuracy of our predictive model and suggests that further experimentation with different architectures of Convolutional neural networks may be fruitful.

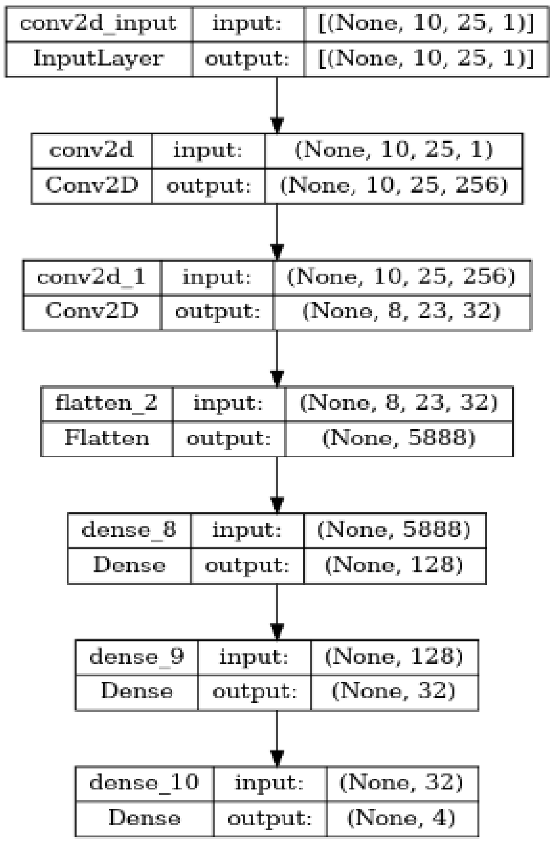
Final Model

2D Convolution: Combinations of Conv2D and Dense Layers

After we interpreted the model based on Conv1D, there seemed to be a strong relationship among the sensor data over time intervals. We formulated an array of dimensions (10, 25) for the reading over the 10 minutes. The output layer had 4 neurons for the one-hot encoded labels. Since the array was reshaped, the dimensions of the input tensor became (35000,10,25,1) [70% of the dataset was used] as 350000 rows and 25 columns are equal to 35000 2d arrays of shape (10,25). After doing hyperparameter tuning, the following characteristics of the model were finalized.

Loss Function	Categorical Cross Entropy
Optimizer	Adam (learning rate = 1e-3)

Outer layer activation function	Softmax
Epochs	100
Batch Size	2048
Validation Split	0.25



conv2d_input | input: | [(None, 10, 25, 1)]
InputLayer | output: | [(None, 10, 25, 1)]
conv2d | input: | (None, 10, 25, 1)
Conv2D | output: | (None, 10, 25, 256)
conv2d_1 | input: | (None, 10, 25, 256)
Conv2D | output: | (None, 8, 23, 32)
flatten_2 | input: | (None, 8, 23, 32)
Flatten | output: | (None, 5888)
dense_8 | input: | (None, 5888)
Dense | output: | (None, 128)
dense_9 | input: | (None, 128)
Dense | output: | (None, 32)
dense_10 | input: | (None, 32)
Dense | output: | (None, 4)

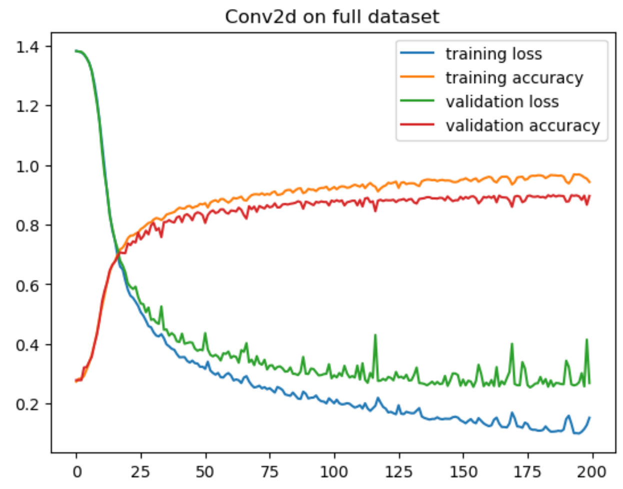
The validation accuracy was calculated on 30% of the dataset. In this case, the validation accuracy reached up to **89%**.

Convolution2D works best as the 2D filters are able to capture the time dependency as well as the sensor dependency for a given kernel size.The moving filter captures this dependency for all sensor and time values. This bi-directional dependency was missing in sequential models like GRU, LSTM.

[For the final submission of the .csv file of the test dataset, we used the whole training dataset for training.]

Experimentation

Hyperparameter tuning



Firstly, we experimented with the kernel size of the filters and the number of filters. Since the dataset was converted to a 2d array of input size(10,25), tweaking the size of the kernel was an important aspect of the training process. When we increased the filters to 256 and set the learning rate to 0.0001, the training accuracy didn't increase abruptly as the epochs progressed, giving better room for the model to generalize. Therefore, we were able to train for several epochs as long as the validation loss decreased. We also tweaked the number of epochs and visualized the best possible epoch for training. We got epoch = 100 to be the best for getting the lowest entropy loss giving us the validation accuracy to be **89%** on the final epoch. In each epoch, the model was validated on 25% of the original dataset when using model.fit(), a method of the TensorFlow library.

Comparison & Results

In our original model which used Random Forest Classifier, our model achieved an accuracy of only 27.3%. This led us to the conclusion that our model was not able to capture the complex relationships between the sensor values. Thus we moved on to more sophisticated models. Applying regular dense layers did not show any significant improvement in accuracy, and we moved to apply RNNs such as GRU and LSTM. However, the model accuracy still remained below 30%, suggesting further improvements in the model.

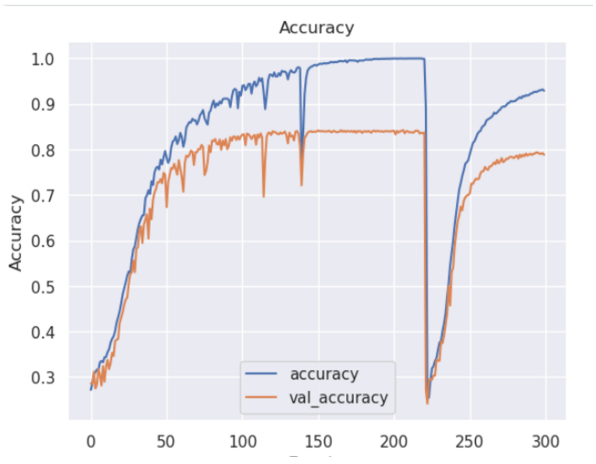
We then applied convolutional layers on LSTM, which showed a drastic increase in accuracy, of 55%. This indicated that convolutional models are able to better capture the data features.

The accuracy jump served as an important motivation and we moved to some heavy applications of convolutional layers. We formulated models consisting of, but not limited to:

- Conv1D + SimpleRNN
- Conv1D + LSTM
- Multiple Conv1Ds
- Multiple Conv2Ds
- Multiple Conv2Ds + Attention
- Conv2D + ResNet

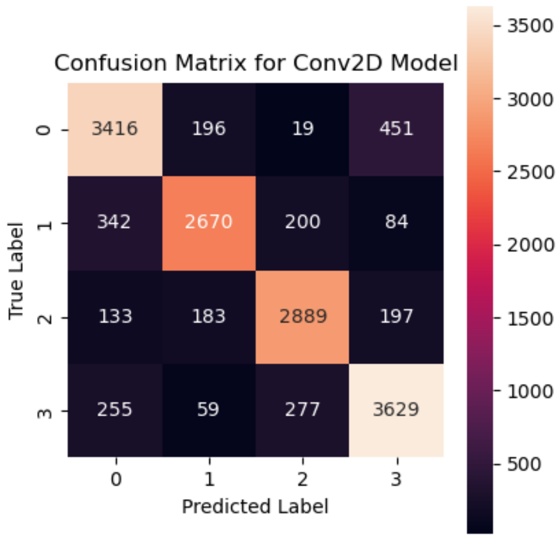
Due to the nature of data, the Conv1D and Conv2D layers performed the best among the rest of the models. LSTM and Attention architectures failed to learn the complexity of the data involved and regular dense layers coupled with Conv1D layers performed enormously better.

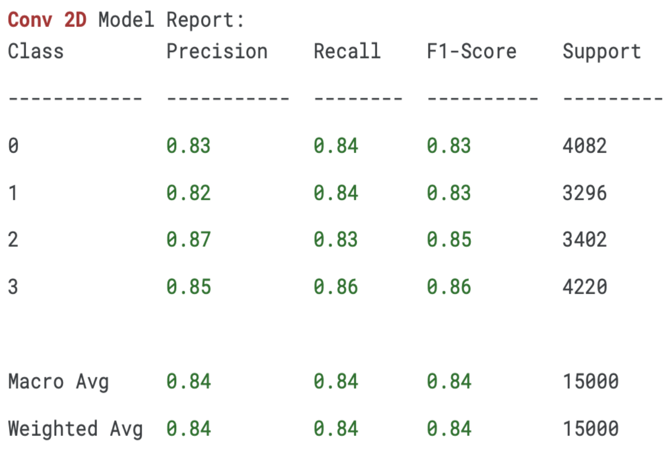
The general intuition of filter size and stride length was unable to capture the variation in data, with stride lengths of 2 and 3 producing unexplainable valleys in accuracy curves.



The final model consisting of 2-D Convolutional layers achieved a peak validation accuracy of 89% as seen from the plot above. The model was tested using 35000 training data points (matrices in this case) which was further split into a 3:1 ratio for validation testing. In this scenario, the confusion matrix obtained from the classification has been plotted below.

The confusion matrix shows 3416 '*No Risk*' labels, 2670 '*Low Risk*' labels, 2889 '*Medium Risk*' labels, and 3629 '*Catastrophic*' labels were correctly predicted by our final model.





```
Conv 2D Model Report:
Class         Precision    Recall    F1-Score    Support

------------  -----------  --------  ----------  ---------

0             0.83         0.84      0.83        4082

1             0.82         0.84      0.83        3296

2             0.87         0.83      0.85        3402

3             0.85         0.86      0.86        4220


Macro Avg     0.84         0.84      0.84        15000

Weighted Avg  0.84         0.84      0.84        15000
```

Model	**Accuracy**
Random forest classifier	27.3%
Support vector machine	28.1%
XG Boost	30.1%
DNN	30%
LSTM + DNN	28.3%
GRU + DNN	28.3%
Convolution 1-D	88.6%
Convolution 1-D with LSTM	55%
Convolution 2-D	**89%**

Conclusion

After trying out different varieties and variations of models, we finally conclude that the Conv2D model stated above is the one which gives the best result and predictions on the provided dataset. With a model accuracy of 89%, we propose this model to be used for industrial purposes, to get the best prediction of the state of the machine during its runtime.

Annexure

References

- Kaggle: Your Home for Data Science
- Keras Conv2D and Convolutional Layers - PyImageSearch
- tf.keras.layers.Conv1D | TensorFlow v2.11.0
- How Do Convolutional Layers Work in Deep Learning Neural Networks? - MachineLearningMastery.com
- SimpleRNN layer (keras.io)
- tf.keras.layers.LSTM | TensorFlow v2.11.0
- Build a multi-class image classification model with the MNIST dataset. | by Shatakshi Singh | Analytics Vidhya | Medium
- Build a GRU RNN in Keras - PythonAlgos
- Keras LSTM Layer Explained for Beginners with Example - MLK - Machine Learning Knowledge