Institut national des sciences appliquées de Toulouse

Génie Mathématiques et Modélisation
Spécialité Mathématiques Appliquées

Project report

Extraction of a flight behaviour from time series data

Authors:
Gonzalez Julie
Roig Lila

Groupe:
MA A
Univeristy year :
2022/2023

Contents

1 Introduction **3**

2 State of the art **3**
- 2.1 Data prepossessing . . . 3
  - 2.1.1 Feature Selection . . . 3
  - 2.1.2 Dimension reduction with Deep Convolutional Autoencoders . . . 4
- 2.2 Multivariate time series classification . . . 5
  - 2.2.1 Short review of time series classification algorithms . . . 5
  - 2.2.2 Multivariate time series classification with Rocket . . . 6
  - 2.2.3 Multivariate time series classification with MiniRocket . . . 8

3 Description of the data **9**

4 Methods **10**
- 4.1 Data preprocessing . . . 10
  - 4.1.1 Identification of qualitative and quantitative variables . . . 10
  - 4.1.2 Separation into training set and test set . . . 10
  - 4.1.3 Feature selection . . . 10
- 4.2 Dimension reduction with 1D-DCAE . . . 11
  - 4.2.1 Preparation of the data . . . 11
  - 4.2.2 Implementation of the 1D-DCAE . . . 12
  - 4.2.3 Evaluation of the DCAE accuracy . . . 13
- 4.3 Multivariate time series classification with MiniRocket . . . 14
  - 4.3.1 Preparation of the data . . . 14
  - 4.3.2 Implementation of MiniRocket . . . 14
  - 4.3.3 Prediction . . . 15
  - 4.3.4 Parameter tuning . . . 15

5 Results **16**
- 5.1 DCAE - Training and accuracy . . . 16
- 5.2 MiniRocket - results . . . 18

6 Conclusion **20**

7 Access to the GitHub repository **20**

1 Introduction

Safety is one of the most important issues in aircraft flight. Flight simulators can be used to reproduce failure scenarios. Simulating flights allows saving fuel and money as it is virtual. Moreover, pilots can practice failure scenarios, which increases their ability to react quickly in emergency situations and take the best decisions in order to increase the safety. Aircrafts are designed so that each system has several back-ups in the event of a failure. Yet, the failure of one component should always be alarming and carefully analyzed in order to reduce the risk of accident.

The data involved in these simulations are time series data, that is to say, they are sequences of points with a temporal structure. A time series is a sequence of point which represents the variations of a parameter through time. A time series is said to be multivariate when multiple parameters vary in parallel, for example here, one time series could describe the variations of the plane's altitude, and one other could be the speed of the plane. More generally, multivariate time series data can be found in many fields, such as economics and finance, or in more recent applications for monitoring systems' behaviours.

Classification is the task of identifying different groups in a set of observations and to assign each observation to a given group. The classification is supervised if the training data given to the classifier have already been labelled, so that we can verify if the algorithm is giving a correct classification or not. In supervised classification task, the parameters of the algorithm are adjusted so that the loss function is minimal. In a classification problem, the loss function counts the proportion of observations that are misclassified.

Classification of time series data is a complex task because the temporal dependencies structuring the time steps of the sequence should be taken into account. It is essential to consider the order of the observation points in a sequence to keep the intrinsic meaning of the time series, and because consecutive points can be highly correlated. Most standard machine learning algorithms must be adapted to be able to treat time series data. In the case of multiple multivariate time series data, the objective is to assign each time series to a given class.

Our aim is to develop an algorithm which will be able to detect the occurrence of a failure and to classify the type of failure played on a flight simulator. Moreover, the algorithm should be able to identify the type of scenario played on the simulator. The algorithms that we are developing are trained on artificial data. However, the final objective is to use these trained algorithms on real data to analyze flights and detect failures in order to increase safety.

After the state of the art gathering the documents on which is based our implementation (section 2), the data is shortly described (section 3). Then, the methods that we implemented for reducing the dimension and performing the classification are explained (section 4). Finally the results are summarized (section 5).

2 State of the art

2.1 Data prepossessing

Time series classification problems can require data preprocessing to select the most important features and reduce the dimensions. For example, [Sodagudi et al., 2022] shows that an excessive dimension of the data can lead to the inaccuracy of the classifier and an excessive increase of the computational complexity. What is more, some features can be redundant or irrelevant, which also leads to an increase of the computational time and increases the risk to over-fit the data.

2.1.1 Feature Selection

A basic approach to reduce the dimension is based on variables' correlation. [Salmina et al., 2021] performs a first feature selection based on the correlation matrix. They aim at identifying failures in the functioning of supermarkets' energy systems in order to optimize their consumption. They work on multivariate time series data. They compute the correlation matrix between all variables

and then identify groups of highly correlated variables. Two variables are considered highly correlated if their correlation is higher than **0.95**. Once highly correlated groups are identified, only one variable per group is kept, after getting the approval of the supermarket. Using this technique, they divided by two the number of variables (from 32 to 16).

2.1.2 Dimension reduction with Deep Convolutional Autoencoders

Deep Convolutional Autoencoders (DCAE) can be used as a data preprocessing for time series classification to reduce the dimension. An autoencoder (AE) is a neural network trained to compress the data in a short code and to decompress this code to reconstruct an output close to the initial data (Cf. [Laurent, 2023]). It is composed of two parts, the encoder which produces the reduced code, and the decoder which decompresses the code, as represented on Figure 1. Thus, the AE's structure provides a mapping between the input data and their reconstruction via an internal sparse representation code. Each component of the encoded vector corresponds to the features learned on the input data. Let's note h the encoder function, g the decoder function and L the loss minimized during the training of the AE. It is interesting to notice that, if h and g are linear and if L is the quadratic loss, then the AE is completely equivalent to a PCA (Principal Component Analysis). PCA is one of the most popular technique for dimension reduction. PCA is a linear technique which consists in projecting the data into sub-spaces where the variance is maximal. On the other hand, if h and g are non linear then the AE is a non linear generalization of the PCA. Thus, AE are adapted for complex data such as time series.

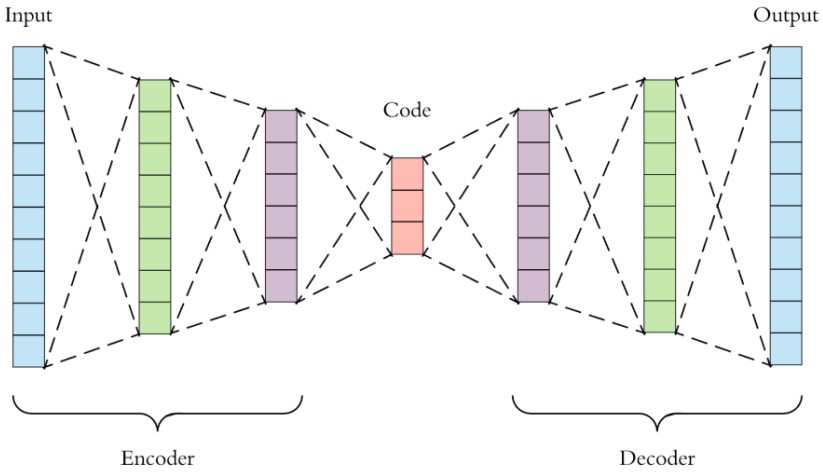

Figure 1: General architecture of an autoencoder (extracted from [Laurent, 2023])

A Deep Convolutional Autoencoder is an autoencoder for which the architecture uses several convolutionnal layers.

[Salmina et al., 2021] used a DCAE in the second step of their data preprocessing, after reducing the number of features as mentioned above. The objective is to reduce the dimension further, not only among the features, but also along the time axis. Before training the DCAE, [Salmina et al., 2021] performs some preprocessing on the data. The first step is to rescale the data. The rescaling is done on each feature separatly, using **min-max normalization**. The formula is given by Equation (1),

$$x'_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{1}$$

where x is one feature vector, x_i is one component of the vector x and x'_i is the component obtained after the min-max normalization. This normalization makes each feature's contribution comparable, that is to say that no feature dominates the others because of its high numerical values. Moreover, re-normalizing can accelerate the convergence of some algorithms. For example, for a neural network, it allows controlling the rate at which the weights of the nodes are

learnt on the training set. The second step is to resample the time series, that is to say, to reduce their resolution in time. To do so, they average the values over one hour, thus each time step of their time series is of one hour. The advantage of resampling is that it reduces the amount of observations. Their time series being two years long, a resolution in hours is still a fine scale. The third step of the preprocessing is to apply a sliding window over the time series. Each sliding window is 168 hours long and is moved with a stride of one hour. Finally, this new dataset is used to train the DCAE. The steps of this implementation are summarized in Figure 2. Later, the classification is performed on the encoded data obtained at the output of the encoder part of the DCAE.

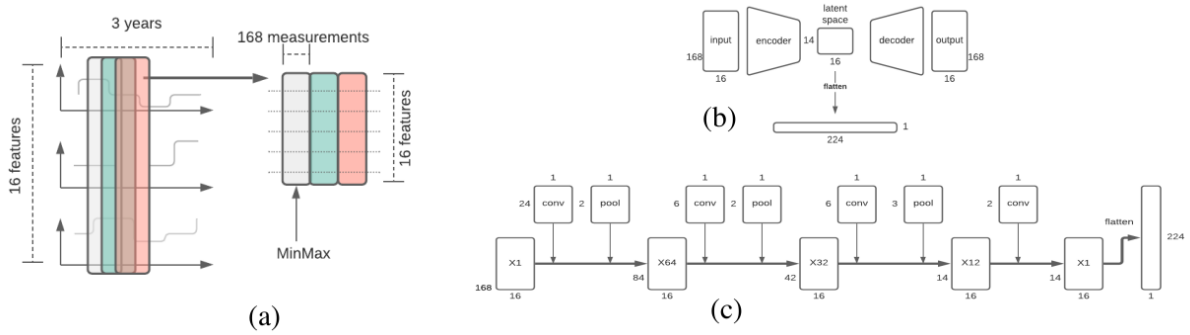

Figure 2: [Salmina et al., 2021] implementation steps for dimension reduction using a DCAE

The use of DCAE as a preprocessing step for time series classification is also exposed in the review [Alqahtani et al., 2021]. As for [Salmina et al., 2021], the preprocessing steps include a feature scaling, followed by a sliding window approach. The justification of the use of a sliding window is exposed. Sliding windows are used to produce a segmentation of the time-series data into shorter segments in time. Sliding windows are appropriate because they provide a preprocessing of raw time series data while preserving their temporal characteristic. Thus, the time series are divided into smaller blocks. [Alqahtani et al., 2021] recommends using overlapping windows so that there is no loss of information. In this way the temporal continuity is kept along the whole time series and there is no fracture in time even though the time series are divided into smaller blocks. To achieve overlapping between the windows, the stride is defined as half the size of the window. Furthermore, [Alqahtani et al., 2021] demonstrates that 1D-DCAE are more appropriate for multivariate time series data because of their high ability to learn complex projection of the data. For example, 1D convolutions are very efficient to discover patterns and features from shorter sequences of the time series. Thus, there are particularly adapted when the data has been preprocessed with the sliding window. 1D-convolutions are similar to dense layers and allow us to obtain very sparse representations. However, the convolution operation has better properties than the flatten operation performed in a dense layer, for example it is robust to scale changes while the flatten operation is not.

2.2 Multivariate time series classification

2.2.1 Short review of time series classification algorithms

Time series classification is a subfield of machine learning with numerous real-life applications. Due to the temporal structure of the data, standard machine learning algorithms are usually not well suited to work on raw time series. Many algorithms have been proposed for multivariate time series classification [Faouzi, 2022a].

Distance-based methods: Nearest-neighbor with dynamic time warping is a simple baseline classification algorithm for time series data. The prediction for a new sample is based on the k labels of samples similar to this new sample. The distance between two time series is calculated with the Dynamic time warping (DTW) metric [Sakoe and Chiba, 1978] which addresses the limitation of the Euclidean distance.

Kernels methods: Support vector machines (SVMs) are supervised algorithms that can be used for classification or regression. In the case of classification, the objective is to find a line that best separates the data in a certain Hilbert space (RKHS), allowing classification even in non-linear cases. SVMs with the global alignment kernel [Cuturi, 2011] perform well for the classification of multivariate time series.

Shapelet-based algorithms: A shapelet is a subsequence of consecutive observations of a time series. Several algorithms rely on shapelets, either by extracting the best shapelets from the training data set [Lines et al., 2012] or by learning them directly [Grabocka et al., 2014].

Tree-based algorithms: Time Series Forest [Deng et al., 2013] trains a Random Forest (RF) algorithm on extracted features from subsequences of time series data. Time series bag-of-feature [Baydogan et al., 2013] is a more advanced algorithm also based on features extracted from subsequences and random forest. This algorithm extracts more features than Time Series Forest. A first RF is trained on a new dataset created from the extracted features and from subsequences of time series data. Then, new features are extracted again and a second RF is trained on them. Proximity forest algorithm [Lucas et al., 2018] works directly on raw time series and with extremely randomized trees algorithm.

Bag-of-words approaches or **dictionary-based approaches**, consists in discretizing time series into sequences of symbols, then extracting words from these sequences with a sliding window, and finally counting the number of words for all the words in the dictionary. These approaches are split into two groups: the ones based on discretizing raw time series such as the SAX-VSM algorithm [Senin and Malinchik, 2013], and the other ones based on discrete Fourier coefficients such as the SFA algorithm [Schäfer and Högqvist, 2012]. Based on the SFA transformation, the Bag-of-SFA-Symbols (BOSS) algorithm was proposed by [Schäfer, 2015]. It uses a representation based on the frequency of occurrence of patterns in time series. Another related method, WEASEL [Schäfer and Leser, 2017], is more accurate than BOSS, but with a similar training complexity and high memory complexity.

Deep learning methods: Over the past decade, deep learning has been investigated for time series classification. InceptionTime [Fawaz et al., 2019] is one of the main deep learning architectures for time series classification. It is a neural network ensemble consisting of five Inception networks. Each Inception module consists of convolutions with kernels of several sizes followed by batch normalization and a rectified linear unit activation function.

Random convolutions algorithms: Convolutional neural networks (CNN) containing several convolutional layers have been investigated for time series classification. For instance, ROCKET algorithm was proposed by [Dempster et al., 2019]. This algorithm extracts features from time series using a large number of random convolutional kernels. These features are then passed to a linear classifier. Several faster extensions have been proposed such as MiniROCKET [Dempster et al., 2021] which reduces the randomness of the parameters.

Ensemble methods, consists of averaging the predictions of several independently trained models into a single prediction in order to build a better final model by decreasing the variance of the predictions. For example, HIVE-COTE [Lines et al., 2016] includes BOSS and Shapelet Transform, as well as classifiers based on elastic distance measures and frequency representations. TS-CHIEF [Shifaz et al., 2019] algorithm is another ensemble model with the same accuracy as HIVE-COTE while having a lower run time.

However, the computational complexity of existing state-of-the-art methods for time series classification makes these methods slow, and impossible to apply to large datasats. More scalable methods include Proximity Forest, TS-CHIEF, InceptionTime and ROCKET.

2.2.2 Multivariate time series classification with Rocket

Most methods for time series classification that attain state-of-the-art accuracy have high computational complexity. Rocket [Dempster et al., 2019] which stands for RandOm Convolutional KErnel Transform is an algorithm for univariate and multivariate time series classification. Rocket uses random convolutional kernels to capture relevant features in time series data and then uses these computed features to train a linear classifier.

Rocket algorithm perform almost equally as recent complex methods such as Proximity Forest, TS-CHIEF or InceptionTime achieving state-of-the-art classification accuracy in a much shorter amount of time. Rocket is more scalable for large datasets, and can be trained on a single CPU core but also on multiple CPU cores of GPUs since it is naturally parallel which further increases its computing speed.

Convolutional neural networks (CNN) are widely used in image classification and can be adapted for time series classification since images and time series have almost the same structure. CNNs use convolutional kernels to detect different patterns in the input. In learning the weights of the kernels, a CNN learns the features associated with different classes present in the data. Each kernel is convolved with the input times series. The output of a convolutional layer is called feature map and can then be used for classification.
The parameters of a kernel are the size, weights, bias, dilation and padding. In time series, kernels are vectors of weights $w_1, ..w_{l_{kernel}}$ that are usually learned during training, where l_{kernel} is the length of the kernel. The bias term b_i is a real number and is added to the result f_i, called feature map, of the convolution between the input data and the kernel k_i. Dilation 'spreads' a kernel over the input and padding appends zero values to the start and end of input time series to extend the input area covered by the kernel as shown in figure 3.

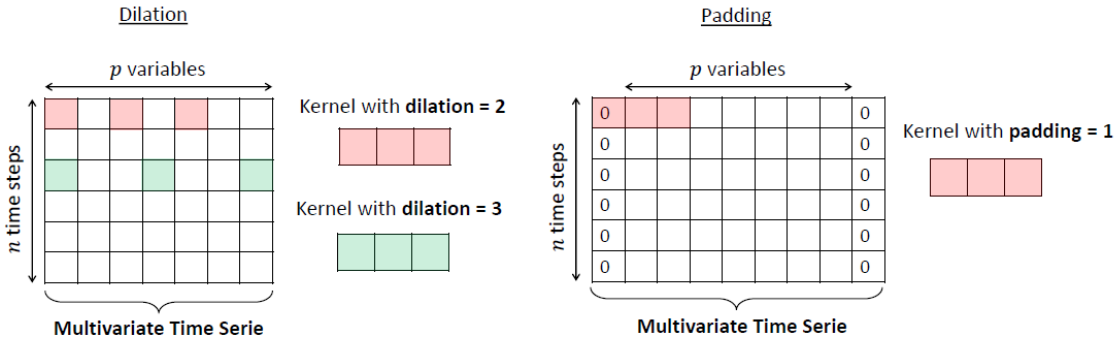

Figure 3: Dilation and Padding mechanisms.

Convolutional kernels can capture many types of features and multiple kernels in combination can capture complex patterns, shapes and frequency information in time series, despite warping.
Kernel weights are learned to represent specific patterns in the input. Thus, the feature map f_i produced in applying a kernel to a time series reflects the extent to which the pattern represented by the kernel is present in the time series. Pooling mechanisms make kernels invariant to the position of patterns in time series. Further, dilation allows kernels with similar weights to capture the same pattern at different scales. Dilation also captures frequency information: larger dilation corresponds to lower frequencies and smaller dilation corresponds to higher frequencies.

Rocket is made of a single convolutional layer with a large number of convolutional kernels. At the end of this convoluted layer, the input time series is transformed into a feature map. Then two features are drawn from this feature map and serve as an input for a Ridge linear classifier. The two extracted features are the maximum value (similar to max pooling) and the proportion of positive values *ppv*. No activation such as ReLU is applied. Figure 4 illustrates Rocket algorithm.
However, in contrast to the learned convolutional kernels that are used in typical CNNs, Rocket uses a large number of **random** convolutional kernels by generating kernels with random length, dilation, padding, weights and bias. The combination of a massive variety of kernels captures features relevant for time series classification, although, in isolation, a single random kernel may not be accurate. As the kernel weights are not learned, and the generation of random kernels is not expensive, the computational cost of the convolutions is low which makes Rocket fast.

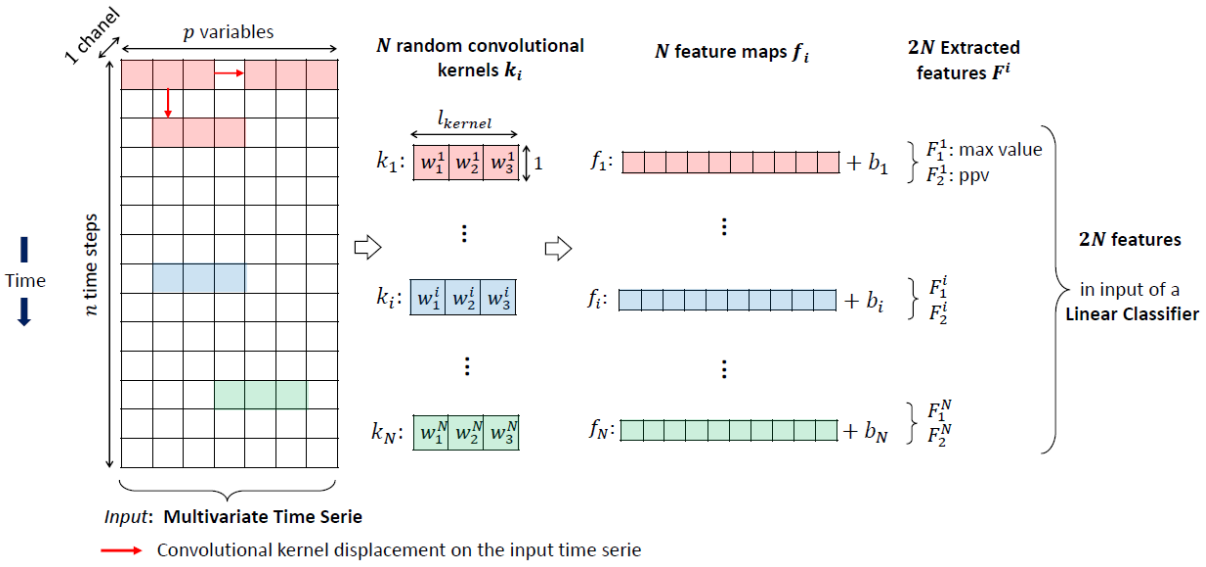

Figure 4: Representation of Rocket algorithm

Therefore, Rocket uses by default:
- $N = 10000$ random convolution kernels producing $2N = 20000$ features as input of the linear classifier.
- Kernels of size 7, 9 or 11 chosen with equal probability.
- Kernel weights sampled from a normal distribution $\forall w \in \mathbf{W}, w \sim \mathcal{N}(0,1)$ and mean centered $w = \mathbf{W} - \overline{\mathbf{W}}$ where $\mathbf{W}$ is the convolution matrix.
- Bias values sampled from a uniform distribution $b \sim \mathcal{U}[-1,1]$.
- Dilation sampled on an exponential scale $d = \lfloor 2^x \rfloor, x \sim \mathcal{U}\left[0, \log_2 \frac{l_{input}-1}{l_{kernel}-1}\right]$ where l_{input} is the length of the time series.
- A padding is applied or not, with equal probability.
- The stride is always 1.

These parameters were determined to produce the highest classification accuracy on a portion of datasets from the UCR archive [Dau et al., 2018] to avoid overfitting on the whole UCR archive, and enabling to generalise well to new problems.

2.2.3 Multivariate time series classification with MiniRocket

MiniRocket algorithm is a reformulation of Rocket seen in section 2.2.2 and stands for MINImally RandOm Convolutional KErnel Transform. MiniRocket has been chosen for this project since it is up to 75 times faster than Rocket on larger datasets, and almost deterministic, while maintaining essentially the same accuracy [Dempster et al., 2021]. Like Rocket, MiniRocket transforms input time series using convolutional kernels, and uses the transformed features to train a linear classifier. However, unlike Rocket, MiniRocket uses a small, fixed set of kernels, and is almost entirely deterministic. Moreover, in contrast to Rocket, it is not necessary to normalize the input time series.

The differences between Rocket and MiniRocket are as follows and summarized in figure 5:
- All kernels k_i are of length 9 with weights restricted to only two values. There are $2^9 = 512$ possible two-valued kernels of length 9 and MiniRocket uses a subset of 84 of these possible kernels.
- Kernel weights are restricted to two values $\alpha = -1$ and $\beta = 2$. The kernels are of the form $k_i = [\alpha, \alpha, \alpha, \beta, \alpha, \alpha, \alpha, \beta, \beta]$, always having 3 values of β. The choice of α and β is arbitrary. The only condition is that the kernel weights sum to zero, ensuring that the convolution output is invariant to the addition or subtraction of any constant value to the input.
- Bias values are computed as follows. For each kernel-dilation combination, a time series is ran-

	ROCKET	MINIROCKET
length	$\{7, 9, 11\}$	9
weights	$\mathcal{N}(0, 1)$	$\{-1, 2\}$
bias	$\mathcal{U}(-1, 1)$	from convolution output
dilation	random	fixed (rel. to input length)
padding	random	fixed
features	PPV + max	PPV
num. features	20K	10K

Figure 5: Summary of changes from Rocket to MiniRocket, from [Dempster et al., 2021]

domly selected and passes through the convolution layer. Then, quantiles (e.g. the $[0.25, 0.5, 0.75]$ quantiles) are drawn from this convolution output and will be taken as possible values for the bias of the given kernel-dilation combination. The random selection of time series for the computation of bias values is the only stochastic element of MiniRocket.
- Dilations are selected in the fixed set $D = \left\{ \lfloor 2^0 \rfloor, ..., \lfloor 2^{\log_2 \frac{l_{input}-1}{l_{kernel}-1}} \rfloor \right\}$. Thus, each kernel is assigned the same fixed set of dilations, adjusted to the length of the input time series. The maximum number of dilations per kernel is set to 32 to keep the transformation efficient.
- Padding is used and alternated for each kernel-dilation combination such that, overall, half the kernel-dilation combinations use padding, and half do not.
- $N = 9996$ convolutional kernels are used (the nearest multiple of 84 less than 10000). MiniRocket only computes *ppv* values and therefore, only $N = 9996$ features are used as the input of the linear classifier.

These parameters are determined to produce the highest classification on 40 UCR datasets with the same aim as Rocket of avoiding overfitting the entire UCR [Dau et al., 2018] archive.

Finally, to significantly speed up the computational time, MiniRocket, computes *ppv* for $\mathbf{W}$ and $-\mathbf{W}$ at the same time, precomputes the product of the kernel weights and the input, and uses these precomputed values to construct the convolution output. Moreover, it reuses the convolution output to compute multiple features, avoids multiplication in the convolution operation and computes all kernels almost at once for each dilation. These computational 'tricks' are made possible by the properties of the small, fixed set of two-valued kernels and of *ppv*.

3 Description of the data

Within the framework of this project, all the datasets were simulated on the flight simulator X-plane. Each simulation gives one multivariate time series output which describes the behaviour of the plane under the played scenario. A time series is a sequence of data points that are indexed by the time. A time series is said to be multivariate when it involves several parameters that evolve in parallel through time. For example, in our problem, the flight is characterized by 4980 parameters that vary throughout the simulation, such as the altitude, the speed, the amount of fuel. Some scenarios do not involve any failures and some lead to specific failure that we aim at identifying. 300 simulations were played and they include 10 types of errors and 5 types of scenarios. In the context of this project, one observation is one multivariate time series, it is a 2D array of dimension the length of the time series times the number of parameters. Thus, the final dataset is 3D dimensional. The time series do not have the same length, they are 40 seconds to 1 minute long. However, they all have a time step of 10 ms. The dataset is composed of **300 observations** and **4980 features**. The fact that there are more features than observations is problematic for most Machine Learning techniques. Thus a preprocessing of the data in order to reduce the dimensionality of the input space is necessary.

4 Methods

4.1 Data preprocessing

4.1.1 Identification of qualitative and quantitative variables

In order to apply machine learning models, it is essential to identify whether the variables are qualitative or quantitative, as the two types of variables do not undergo the same preprocessing and are not treated in the same way by the algorithms. However, as there are 4980 parameters, this identification work cannot be done entirely by hand as it would be too time consuming and tedious. The company SII provided us with an excel table containing the unit and type of each parameter. However, it is not possible to use the type of parameters to determine whether they are qualitative or quantitative, since the booleans, for example, are encoded with integers (quantitative), whereas they are qualitative parameters. We therefore manually review each unit (e.g. seconds, kgs, amps, bitfield, bool...) to determine whether the parameters that correspond to that unit are quantitative and qualitative. We assign with a function the parameters having quantitative units as quantitative, and the parameters having qualitative units as qualitative.

Some units such as EPR or degm are unknown to us and are not explained in the excel. In this case, we looked at the values of the associated parameters to determine their category. The variables associated with an unknown type '???' were also processed manually.

Subsequently, some parameters are in array form while others are scalars. For a given parameter in an array form, we consider that each column of the array is a variable in itself. Thus, we created a procedure to identify each column of the table as a quantitative or qualitative variable. Finally, we saved the created lists containing the quantitative and qualitative variables.

4.1.2 Separation into training set and test set

The dataset is separated into two subsets. The training set is used to train the classification algorithm. The test set is used to measure the accuracy of the classification model, as the data are labeled. SII initially provided us with a 50/50 data separation with 50% of the data in the train set and 50% of the data in the test set. The split was not done randomly. We first trained our algorithms on this data separation as we then have each failure scenario in both training and test sets. However, the fact that the data were not separated randomly can introduce a bias in the classification. Moreover, the dataset is quite small, there are only 300 time series in total, thus with this 50/50 separation there are only 150 time series to train the algorithms. To avoid these problems, we proposed a 80/20 data separation with 80% of the data in the training set and 20% of the data in the test set. The split between the test set and the training set was done randomly so that no bias is introduced in the classification. Moreover, the data was shuffled because in the initial dataset the scenarios were ordered with all the normal scenarios first and then all the failure scenarios. The algorithms were tested with both configurations.

4.1.3 Feature selection

In our dataset, the number of features is about 16 times more important than the number of observations. Thus, following [Faouzi, 2022b]'s recommendations to improve time series classification, we first perform a feature selection to reduce the dimension. This feature selection is performed on the training set only. Once the features are selected, the test set is modified and only the features that were selected on the training data are kept.

First, an analysis of the data showed that many variables are constant and equal between all the scenarios. Thus, these variables are irrelevant as they do not bring any information regarding possible failures. We delete them to reduce the dimension. Moreover, some of the plane's variables correspond to warnings that are triggered by failures. These variables have to be deleted otherwise they would interfere with the failure classifier and give it the label of the observations.

Following [Salmina et al., 2021]'s approach, we perform a first feature selection based on the computation of the correlation matrix. First, the correlation matrix of the variables is computed. Then, the groups of variables that have a correlation greater than **0.95** are identified and in each of these groups only one variable is kept. Contrary to [Salmina et al., 2021], we do not have any ground truth knowledge about the plane parameters involved in the failures, so we randomly select the variable kept in each highly correlated group. Moreover, the dataset includes both qualitative and quantitative variables that must be treated separately regarding the correlation. The correlation between qualitative variables is handled by a χ^2 test performed with the library *dython*. For the quantitative variables, the correlation matrix is computed using the library *pandas*. Only the variables that are highly correlated for all time series are deleted. The library *dython* also offers the possibility to compute a correlation matrix on a dataset containing both qualitative and quantitative variables, but the computations are very heavy. Furthermore, the first step which consists in deleting constant and equal variables among all time series appears to be quite essential to reduce the dimension in a light computational way. In this way, when the correlations are computed the dimension has already been reduced. This makes the computations lighter because computing the correlation matrices is more expensive than searching for constant and equal time series. To conclude, after this feature selection, **125 quantitative variables** and **79 qualitative variables are kept**, so the dimension has already been considerably reduced.

4.2 Dimension reduction with 1D-DCAE

Based on the publications [Alqahtani et al., 2021] and [Salmina et al., 2021], we implement a 1D-DCAE for the dimension reduction of the multivariate time series data.

4.2.1 Preparation of the data

First, we prepare the data so that it is adapted to be treated by the DCAE. First of all, our dataset is composed of multiple multivariate time series data. The different time series do not have the same length so their lengths have to be equalized. One way to do so is to take the length of the longest time series as a reference. Then, the time series that are shorter are completed with zeros until the length of the resulting time series equals the reference length. The fact that the same value is repeated at the end does not alter the information contained in the time series because it does not create any new variations in the time series. It is necessary that all multivariate time series have the same length because the dimension of the input layer of the 1D-DCAE will be fixed.

Next, we rescale the data using a **min-max normalization** as described in Equation (1). It is important to notice that the rescaling should be done after the extension of the time series length. Otherwise, the longest time series, which does not need to be extended, would have a different scaling than the shorter time series.

Then, we resample the data to make a first reduction of the time dimension. Initially the time series have a resolution of 10 ms. We reduce the **resolution up to 25 ms**. In this way, the amount of data points in the time series is divided by 2.5 and a resolution of 25 ms seems reasonable enough for the detection of anomalies, knowing that the time series are about 1 minute and 40 seconds long.

Next, a sliding window is performed on the data. The two important parameters to fix are the window size and the stride. As described by [Alqahtani et al., 2021], the window is fixed to be twice the size of the stride, so that windows are overlapping. It is important to notice that, depending on the stride we choose, we may leave out a more or less important part of the time series at the end of the sliding window. For example, let's assume the length of the time series is 110 seconds and the chosen stride is 20 seconds. Then, the window length will be 40 seconds, so the last time the window can be slid is at 60 seconds of the time series and it will cover the time series until 100 seconds. Thus, in this case, the last 10 seconds will be left out from the data. The bigger the stride the more risks there are to leave out an important amount of data.

However, if the stride is very small the amount of data generated by the slid window will be very large and the obtained sub parts of the time series will be too short. Thus, we want to define a stride that is large enough but for which we do not leave out a too long part of the end of the time series. Regarding these criteria, we define the maximal stride s_{max} as being 5% of the total length of the time series. For example, for a time series of 100 seconds, the maximal stride is 5 seconds. Then, the chosen stride is defined as the largest value in the interval $[1, s_{max}]$ such that the proportion $p = \frac{\text{duration of the time series that is left out}}{\text{size of the slid window}}$ is lower than 0.02. In this way, we do not leave out more than 2% of the size of the slid window. At the end of the sliding window step, each multivariate time series is associated to a 3D tensor of dimension (length of the time series $\times$ number of features $\times$ number of slid windows), as represented on Figure 6.

Finally, the last step of the preprocessing is to arange the data in one unique 4D tensor, so that we can give it as an input to the 1D-DCAE. The dataset contains multiple multivariate time series, and each of them is now associated to a 3D tensor. So, the resulting tensor is 4 dimensional, of shape (number of multivariate time series $\times$ size of the window $\times$ number of features $\times$ number of slid windows) as represented on Figure 7.

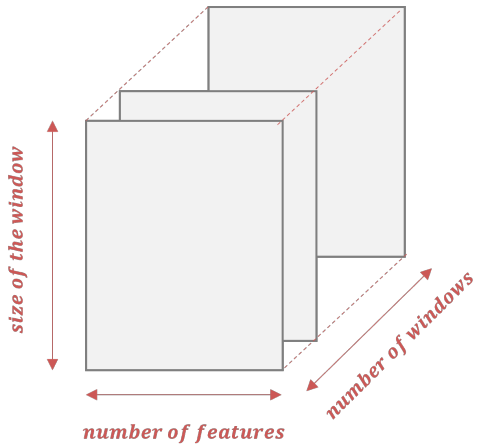

Figure 6: Data obtained after the sliding window approach on one time series

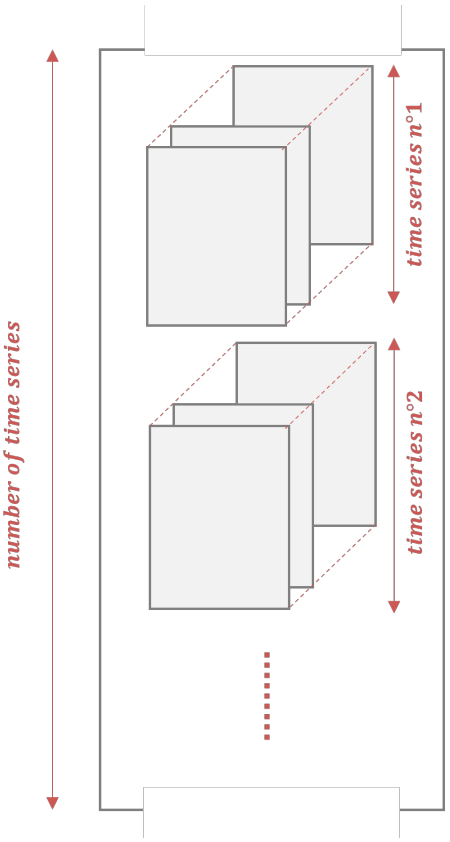

Figure 7: 4D input data of the 1D-DCAE

4.2.2 Implementation of the 1D-DCAE

We implemented two different architectures of the 1D-DCAE The first architecture is based on the architecture proposed by [Alqahtani et al., 2021]. The architecture of the encoder part of the network starts with four consecutive 1D-convolutional layers, followed by two fully connected layers. All layers have a ReLU activation function. The first three 1D-convolutional layers have a filter size of 10×1, and the fourth layer has a filter size of 1×3. There are 32 kernels in the first 1D-convolutional layer, 64 in the second, 128 in the third and 128 in the fourth. All convolutional layers include a padding of 1. Following the convolutional layers, the first fully connected layer has 384 kernels and the second one has 5 kernels. This second fully connected layer gives the sparse encoded data. At the output of this layer starts the decoder block. The decoder's architecture mirrors the one of the encoder. Figure 8 summarizes the architecture of

the 1D-DCAE.

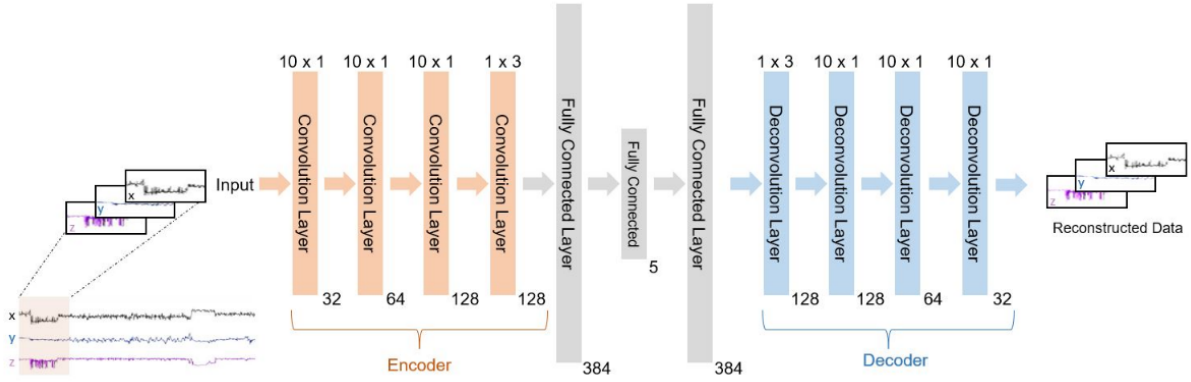

Figure 8: Architecture of the 1D-DCAE implemented by [Alqahtani et al., 2021]

The most important adaptation that had to be brought to this architecture was to modify the shape of the input layer of the encoder and decoder, so that they would accept 4 dimensional tensors. The shape of the input data is defined as (size of the window $\times$ number of features $\times$ number of slid windows). Thus, the number of kernels on the input layer is no more 32 in our adaptation of the architecture, it has to be equal to the number of slid windows (it represents the number of channels). In the same way, the number of neurons in the output layer of the decoder is equal to the number of slid windows in order to reconstruct an output of the same size as the input. With this first architecture the dimension reduction is obtained row-wise, so only the time steps are being reduced.

To see the impact of reducing both the time and the number of features, a second architecture was implemented. This architecture combines the concepts of [Salmina et al., 2021] and [Alqahtani et al., 2021]. The main difference between these two architectures is that [Salmina et al., 2021] performs **Maxpooling** during the encoding part, and then **Upsampling** in the decoding part, while [Alqahtani et al., 2021] does not. Applying Maxpooling with a filter of dimension $(1, f)$ allows dimension reduction column-wise. The purpose of this layer is to reduce the size of the data without modifying their additional features. The Maxpooling operation consists in applying a filter of size $(1, f)$ to each sub-part of the data and for each application, the filter sends back the highest value of the sub-part. As an autoencoder aims at recreating its input, to counteract the dimension reduction by the Maxpooling layer in the encoder, the decoder includes an **Upsampling** layer of size $(1, f)$, which brings back the original dimension. Thus, the second implemented architecture includes one Maxpooling layer, with filter size $(1, 2)$, placed between the first and the second convolutional layer of the encoder part of the architecture described by Figure 8. In addition to that, the decoder part takes one new Upsampling layer, with filter size $(1, 2)$, between the penultimate and last layers.

4.2.3 Evaluation of the DCAE accuracy

We used the following metrics to evaluate the DCAE model :

- **R^2** represents the proportion of variance explained by the model, R^2 varies from 0 (the model does not fit the data) to 1 (all the variance is explained):

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2} \tag{2}$$

- **RMSE** (root-mean-square error) assesses the prediction performance of the model and

strongly penalizes outliers, the lower the RMSE the better the prediction:

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3}$$

- **MAE** (mean-absolute error) is the average of residuals, it allows to control if the model represents well the main trend of the data. The lower the MAE the better the prediction:

$$MAE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4}$$

4.3 Multivariate time series classification with MiniRocket

4.3.1 Preparation of the data

As mentioned above, by applying MiniRocket to the multivariate time series provided by SII, we want to perform three different classifications. The first classification is binary and aims to detect the occurrence of a failure during the flight. The values taken by the target variable is 1 if a failure occurred or 0 otherwise. The second objective is to perform multiclass classification to identity the type of flight scenario. The 5 scenarios taken by the target variable are the plane's take-off, and 4 different versions of the plane's take-off and a turn. Finally, the last task is also a multiclass classification to identify the type of failure that occurred during the flight where the target value can take 11 different failure scenarios. We create a function to retrieve the array of labels for each classification task. Thus, throughout this section, the codes will be applied for the three classification tasks as well as for the two datasets configuration (train/test separation 80/20 and 50/50).

For a given classification task and dataset separation, we start by loading the time series data. We first load the data reduced by the DCAE as described in the section 4.2. In order to compare the results, we also load the 'raw' data that have not been transformed by sliding windows nor reduced by the DCAE. However, this data have undergone the same preprocessing as in section 4.2.1 that is length equalisation, min-max normalization and resolution reduction.

4.3.2 Implementation of MiniRocket

The original implementation of MiniRocket is based on the code [Dempster et al., 2022] associated with the paper [Dempster et al., 2021]. This implementation uses the Python library *numba* to translate a subset of Python and NumPy code into fast machine code. A GPU implementation of MiniRocket running 3 to 25 times faster is available through the Python library *tsai*. *tsai* is an open-source deep learning package built on top of Pytorch providing state-of-the-art techniques for time series tasks. We will also use *fastai* Python library which simplifies the training of neural networks. The authors provide with this *tsai* code a tutorial [Malcom and Ignacio, 2022] on which we based our implementation of MiniRocket using google colab GPUs.

The MiniRocket implementation requires the data to be in the form of 2D numpy arrays. However, each time series output from the DCAE results from the application of sliding windows and is therefore a 3D matrix of size (window size $\times$ number of features in the time series $\times$ number of windows). We juxtapose the sliding windows from the DCAE one after the other to recreate a 2D structure.

All features obtained with MiniRocket are calculated in a first stage with the function *MiniRocketFeatures* from *tsai* library. *MiniRocketFeatures* is fitted only on the training data and the features obtained will remain the same throughout training. Once the features are computed, they are passed to a *tsai* dataloader that will create batches. Then, a linear classifier head is added to the model with the function *MiniRocketHead*. It is made of a flatten *pytorch*

layer to reshape the input into a 1D tensor, a batch normalization layer and a linear layer with a dropout parameter. Dropout is a technique for reducing overfitting during model training by randomly removing neurons in the linear layer with a certain probability p. To train the model, we use the *fit_one_cycle* function from *fastai* which uses large, cyclical learning rates to train models significantly quicker and with higher accuracy than just using the *fit* method.

4.3.3 Prediction

Once the features are computed on the train set and the model is trained with these features, we implement a function to perform the prediction on the test data. To do this, we generate new features for the test set using the *MiniRocketFeatures* function that was fitted on the train set. We use the *fastai* function *get_X_preds* to compute the predictions on the test set.

To evaluate the performance of the model, the confusion matrices are plotted and the accuracy and F1-score metrics from *scikit learn* library are used. A perfect model has an accuracy and a F1-score of 1.

- The **accuracy** is the fraction of predictions correctly identified by our model.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{5}$$

- The **F1-score** can also measure of a model's accuracy on a dataset. Is an harmonic mean between the precision and the recall of the model, and is particularly used in the case of imbalanced data.

$$\text{F1-score} = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{6}$$

where

$$\text{precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \qquad \text{recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

4.3.4 Parameter tuning

We aim to tune three hyper-parameters: the dropout value of the linear classifier, the learning rate value and the number of epochs with which the model is trained. First, we tune the hyper-parameters by hand using the *lr_find* function of *fastai*. *lr_find* plots the loss as a function of the learning rate. To reduce the amount of guesswork on picking a good starting learning rate when training the model, we choose a value that is approximately in the middle of the sharpest downward slope on the plotted figure as shown in figure 9.

However, tuning the parameters by hand may not be optimal. In order to optimize the tuning and find the parameters giving the best possible classification accuracy, we implement a grid search function, as no such function is already available with *tsai*.

To do this, we first implement a function to calculate the accuracy of a model by performing a k-fold cross-validation. For the k-fold cross-validation algorithm, the training data set is divided into k folds. During each iteration, the model uses one of the k folds to test the model and calculate the accuracy of the prediction. The other remaining folds are used to train the model. The same process is repeated for all k folds. The final accuracy returned by this function is the average of the accuracy of each fold. This method allows us to evaluate the model without using the test set, while providing a good approximation of the model's accuracy. Thus, we can test our model for several parameters (i.e. perform model selection) without touching the test set.

Thereafter, we recall that we want to tune the 3 parameters: dropout, learning rate and number of epochs. For this, we need to evaluate our model for several possible configurations of these 3 parameters. As we cannot test all configurations and because each run is time consuming, we

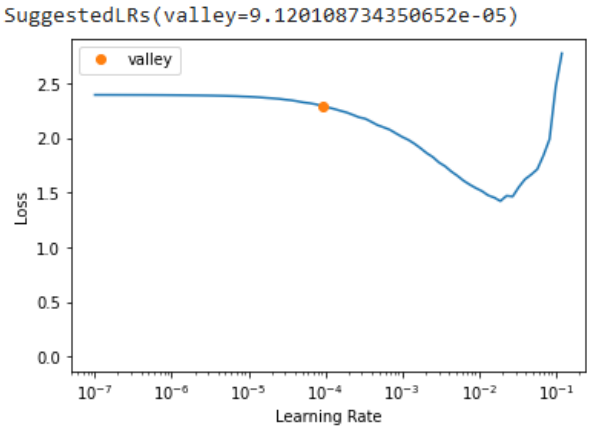

Figure 9: Plot produced by *lr_find*. A good choice for the learning rate value would be 10^{-3}.

want to cover as many parameter values as possible with a minimal number of model evaluations. To do this, we implement a function that creates a grid of values for the parameters in order to optimally cover the space of possible values. This function uses the *LatinHypercube* function of *scipy* library to perform Latin Hypercube Sampling (LHS). LHS is a way of generating random samples of parameter values. As we wish to tune 3 parameters, the space of possible values for the parameters is included in a 3-dimensional hypercube. A Latin Hypercube (LH) consists of splitting the 3D hypercube into equal partitions and ensuring only one parameter value in each partition. We generate $N = 30$ LH and take as the final grid of values, the LH that minimizes the discrepancy criteria i.e. that best covers the parameters space.

Finally, we implement our grid search function, taking as input the grid of values given by the best LH and performing a k-fold cross-validation for each parameter configuration of the grid. The configuration giving the best cross-validated accuracy is selected.

It turns out that performing parameter tuning on the 'raw' dataset (without going through the DCAE) is too time consuming. We therefore perform the parameter tuning on the reduced data from the DCAE to find the best parameters. Then, we use the best parameters found to train the model on the 'raw' data and on the data reduced by the DCAE.

5 Results

5.1 DCAE - Training and accuracy

The DCAE is trained on the training dataset to compute the optimal weights of the neurons. Throughout the training the loss that is minimized is the binary cross entropy from the *Tensorflow* library. The optimizer used by the optimization algorithm for the minimization is Adam from the *Tensorflow* library. The DCAE was trained with batches of size 10 on 30 epochs for the data separation 50/50 and 20 epochs for the data separation 80/20.

The number of epochs had to be calibrated so that the algorithm would converge but that the DCAE would not overfit the data. In fact, if the number of epochs is too large, the DCAE can overfit the training set, which leads to bad generalization properties. To verify that the number of epochs is reasonable, the decreasing loss is plotted through the training (Figure 10). The algorithm converges when the loss stabilizes on a low and constant value. For both architectures, the loss decreases exponentially and seems stable after 15 iterations on the dataset 80/20 and after 22 iterations for the dataset 50/50. Moreover, for the first architecture, the loss stabilizes around 0.26, while for the second architecture it stabilizes around 0.37. This suggests that the first architecture fits the training data better than the second one, even though both have converged. This result makes sense because in the second architecture, the dimension of

the encoded data is twice smaller than for the first architecture, so reconstructing the data is more difficult.

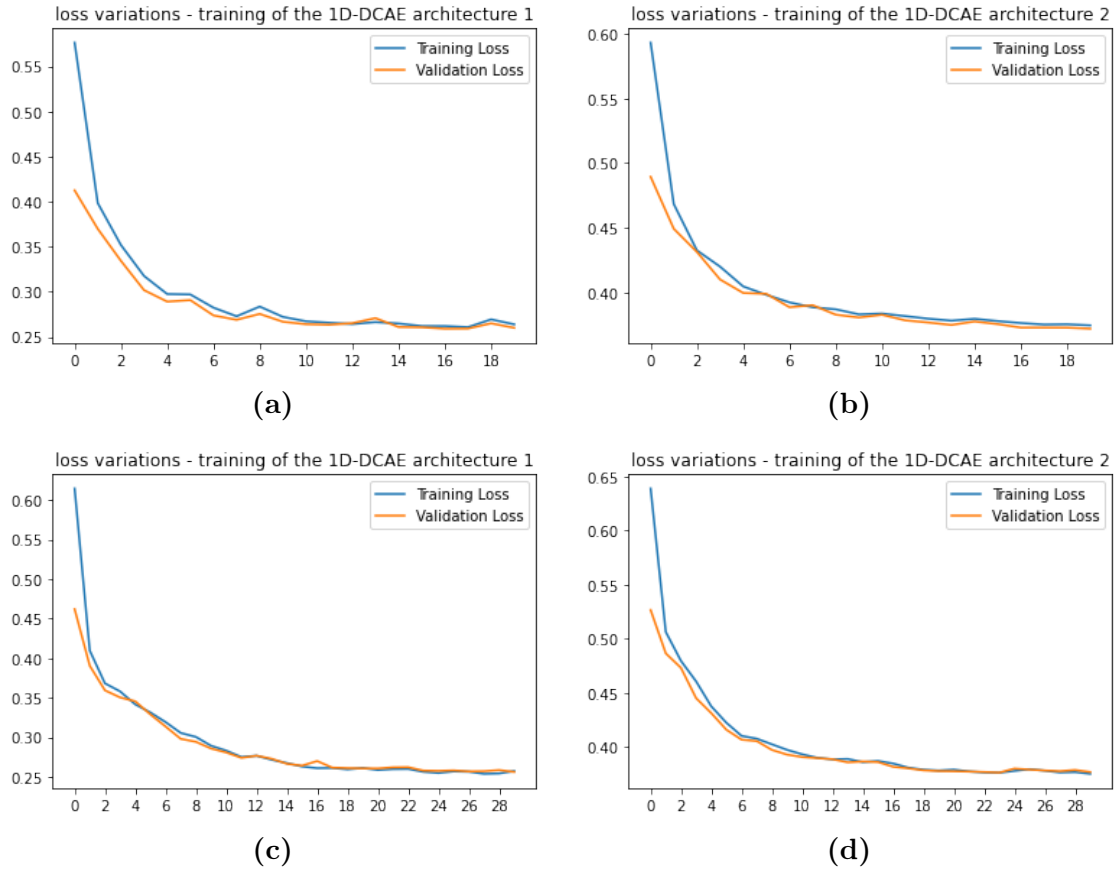

Figure 10: Loss function **(a)** architecture 1, data separation 80/20 **(b)** architecture 2, data separation 80/20 **(c)** architecture 1, data separation 50/50 **(c)** architecture 2, data separation 50/50

After training, the DCAE is used to predict the test set in order to compute evaluation metrics and measure the performance of the model. Table 1 summarizes the score obtained with different metrics on the two architectures.

Table 1: Summary of all models' results

					evaluation metrics		
Architectures	Data separation	Epochs	Batch size	Loss value	RMSE	MAE	R^2
Architecture 1	50/50	30	10	0.25	0.089	0.029	0.97
Architecture 2	50/50	30	10	0.38	0.22	0.14	0.68
Architecture 1	80/20	20	10	0.26	0.072	0.027	0.97
Architecture 2	80/20	20	10	0.37	0.22	0.14	0.69

The obtained scores confirm that the first architecture reconstructed the data better than the second. This gives us an indication about the amount of data that is used when making a dimension reduction with the DCAE. However, this result gives no indication on how well the classification algorithm will perform on the data provided by the DCAE. For example, even if the first architecture has a better accuracy, the classification might perform better on the second architecture as the data's dimension has been reduced more consistently.

5.2 MiniRocket - results

We perform several runs of the model for different possible configurations: 80/20 and 50/50 data separation, the three classification tasks (error detection, type of scenario, type of error), 'raw' data or data reduced by the DCAE and parameters tuned by hand or by a grid search. The results are shown in table 2 and 3. The best results obtained for each classification task are highlighted in bold in the tables. For the dataset 80/20, the best results for the detection of the error and the type of error were obtained on the reduced data provided by the first DCAE architecture. The error was detected with an accuracy of 0.9 and a F1-score of 0.867. The type of error was identified with an accuracy of 0.85 and a F1-score of 0.549. Finally, the best result for the recognition of the type of scenario was obtained on the raw data, with an accuracy of 0.883 and a F1-score of 0.839. For the dataset 50/50, the best results for the detection of the error and the type of error were obtained on the reduced data provided by the second DCAE architecture. The error was detected with an accuracy of 0.893 and a F1-score of 0.868. The type of error was identified with an accuracy of 0.8 and a F1-score of 0.461. Finally, the best result for the recognition of the type of scenario was obtained on the raw data, with an accuracy of 0.853 and a F1-score of 0.846.

DCAE output	Classification task	Parameter tuning	Learning rate	Number of epochs	Dropout	Accuracy	F1-score
architecture 1 20 epochs	error detection	by hand	10^{-3}	5	0.2	0.883	0.841
		grid search	8.23×10^{-4}	18	0.36	0.833	0.795
architecture 1 20 epochs	type of scenario	by hand	10^{-3}	10	0.2	0.783	0.719
		grid search	8.40×10^{-3}	19	0.21	0.733	0.645
architecture 1 20 epochs	type of error	by hand	10^{-3}	9	0.2	0.8	0.372
		grid search	5.20×10^{-3}	18	0.50	0.85	0.51
architecture 1 30 epochs	**error detection**	**by hand**	10^{-3}	**8**	**0.2**	**0.9**	**0.867**
		grid search	2.84×10^{-3}	18	0.013	0.867	0.822
architecture 1 30 epochs	type of scenario	by hand	10^{-3}	10	0.5	0.75	0.684
		grid search	3.10×10^{-3}	13	0.03	0.733	0.641
architecture 1 30 epochs	**type of error**	**by hand**	10^{-3}	**10**	**0.5**	**0.85**	**0.549**
		grid search	1.50×10^{-3}	16	0.38	0.833	0.474
architecture 2 20 epochs	error detection	by hand	10^{-3}	10	0.3	0.883	0.841
		grid search	8.45×10^{-3}	19	0.39	0.9	0.872
architecture 2 20 epochs	type of scenario	by hand	10^{-3}	10	0.5	0.783	0.715
		grid search	2.14×10^{-3}	19	0.062	0.817	0.731
architecture 2 20 epochs	type of error	by hand	10^{-3}	10	0.2	0.8	0.577
		grid search	3.74×10^{-3}	15	0.37	0.833	0.548
raw data	error detection	by hand	10^{-3}	10	0.2	0.833	0.778
		grid search	8.45×10^{-3}	19	0.39	0.817	0.778
raw data	**type of scenario**	**by hand**	10^{-2}	**8**	**0.5**	**0.883**	**0.839**
		grid search	2.14×10^{-3}	19	0.062	0.867	0.851
raw data	type of error	by hand	10^{-3}	10	0.2	0.833	0.561
		grid search	5.20×10^{-3}	18	0.50	0.8	0.497

Table 2: Results obtained on the dataset with the separation 80/20. The Accuracy and the F1-score are computed on the test set

For both datasets configurations, we observe that the DCAE improves the results in the case of the classification tasks *error detection* and *type of error*. The best results are obtained for binary classification *error detection*, while the worst results are obtained with multiclass classification *type of error*. The more classes there are, the more the accuracy decreases. Moreover, grid search tuning did not improve the results compared to hand tuning. This could be explained by the fact that we did not create a sufficiently detailed mesh for the grid of values (noting that the more detailed the mesh, the more parameters to be tested and the longer the calculation times) and by the fact that that we performed a very large number of tests for the tuning of the parameters by hand. Thereafter, we notice that for the 50/50 dataset, the algorithm has to be trained with more epochs than when training with the 80/20 dataset. This could be explained by the fact that the 50/50 dataset has less training data than the 80/20 dataset. In addition, the 50/50 dataset has a larger and therefore more diverse test set than the 80/20 dataset. This

could explain why we get slightly worse results with the 50/50 dataset. Finally, the F1-score in the case of the classification task *type of error* is very low in both datasets. This is explained by the fact that for this classification task, the classes are unbalanced with a predominance of the 'None' class, leading to a low F1-score. The confusion matrices given in figure 11 illustrate this phenomenon.

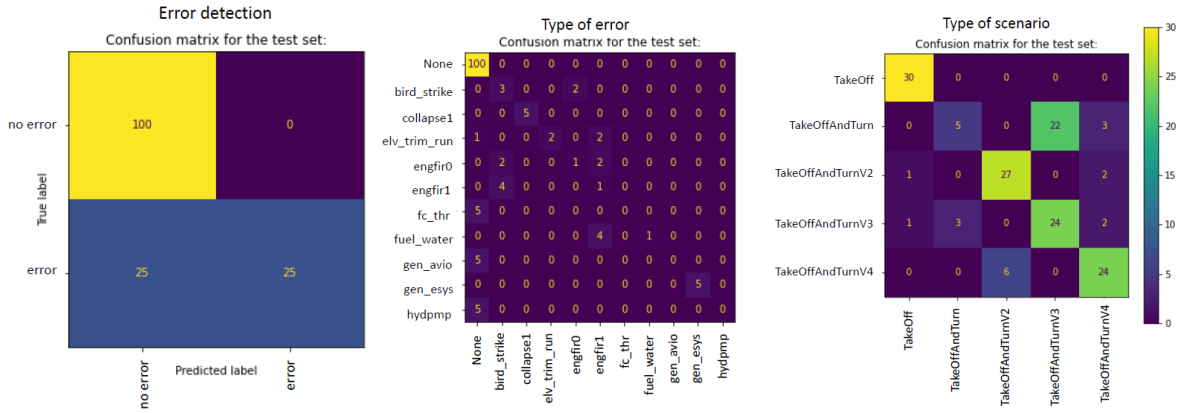

Figure 11: Confusion Matrices obtained with setup: dataset 50/50, DCAE Architecture 2 trained on 20 epochs.

DCAE output	Classification task	Parameter tuning	Learning rate	Number of epochs	Dropout	Accuracy	F1-score
architecture 1 20 epochs	error detection	by hand	10^{-3}	20	0.2	0.887	0.858
		grid search	2.55×10^{-4}	16	0.35	0.88	0.851
architecture 1 20 epochs	type of scenario	by hand	0.002	20	0.2	0.8	0.789
		grid search	7.00×10^{-3}	19	0.05	0.747	0.715
architecture 1 20 epochs	type of error	by hand	10^{-3}	20	0.3	0.733	0.314
		grid search	2.58×10^{-3}	17	0.41	0.76	0.37
architecture 1 30 epochs	error detection	by hand	0.01	12	0.2	0.827	0.785
		grid search	1.40×10^{-3}	17	0.31	0.853	0.815
architecture 1 30 epochs	type of scenario	by hand	10^{-3}	20	0.5	0.747	0.722
		grid search	9.06×10^{-3}	16	0.17	0.693	0.681
architecture 1 30 epochs	type of error	by hand	10^{-3}	20	0.2	0.713	0.281
		grid search	1.39×10^{-3}	20	0.11	0.773	0.395
architecture 2 20 epochs	**error detection**	**by hand**	10^{-3}	**20**	**0.2**	**0.893**	**0.868**
		grid search	2.86×10^{-3}	12	0.20	0.833	0.778
architecture 2 20 epochs	type of scenario	by hand	10^{-3}	20	0.5	0.747	0.718
		grid search	1.12×10^{-3}	16	0.17	0.733	0.701
architecture 2 20 epochs	**type of error**	**by hand**	10^{-3}	**20**	**0.5**	**0.8**	**0.461**
		grid search	1.10×10^{-3}	16	0.47	0.787	0.423
raw data	error detection	by hand	10^{-3}	12	0.5	0.84	0.789
		grid search	2.55×10^{-4}	16	0.35	0.86	0.822
raw data	**type of scenario**	**by hand**	3×10^{-3}	**20**	**0.5**	**0.853**	**0.846**
		grid search	7.00×10^{-3}	19	0.05	0.747	0.339
raw data	type of error	by hand	2×10^{-3}	20	0.3	0.74	0.355
		grid search	1.10×10^{-3}	16	0.47	0.76	0.374

Table 3: Results obtained on the dataset with the separation 50/50. The Accuracy and the F1-score are computed on the test set

6 Conclusion

We implemented a Deep Learning method to detect the occurrence of a failure and to identify the type of failure and the type of scenario played on the simulator. As the data are multivariate time series all the implemented method must be adapted and must take into account the temporal dependencies structuring the time steps.

As there are about 5,000 parameters, we first performed a feature selection. The parameters that are constant through time and between all time series were deleted. Moreover, for each group of highly correlated variables only one variable was kept. This feature selection is necessary so that the classification algorithm is able to handle the data. Second, a dimension reduction was performed by a Deep Convolutional Autoencoder (DCAE). Two architectures were proposed and implemented, the first one offering only a reduction timewise, and the second one offering in addition a reduction of the features. Then, the algorithm MiniRocket performed a supervised classification on the reduced data to identify the occurrence of a failure, classify the type of failure and the type of scenario. The classification is supervised because the training data given to the classifier are labelled to indicate if there is a failure in the played scenario, the type of failure and the type of scenario. In this way, the classification algorithm is trained on the training data with the real output classification. Later on, when new unlabelled data are introduced for testing, the algorithm is then able to predict their class.

When comparing the results of the two DCAE architectures on the reconstruction of the test set, the first architecture obtained better results, which is consistent as the data is less reduced. However, the second architecture reduces the data more, thus the training of the algorithm MiniRocket is faster. Then, the algorithm MiniRocket was trained on the data reduced by the DCAE and on the raw data preprocessed but not reduced by the DCAE. This allowed us to evaluate if the dimension reduction provided by the DCAE improved our results and which of the two architectures provided the best results.

For the identification of the type of scenario, the best results were obtained for the classification of the raw data on both datasets 80/20 and 50/50. We obtained respectively an accuracy of 88.3% and 85.3%. For the detection of the occurrence of a failure, the best results were obtained on the data reduced with the first DCAE architecture for the dataset 80/20, and on the data reduced with the second DCAE architecture for the dataset 50/50. For both, the obtained accuracy was about 90%. Finally, the best results for the identification of the type of error were also obtained on reduced data. The accuracy was 85% on the data reduced with the first DCAE architecture and 89% on the data reduced with the second DCAE architecture.

To conclude, the dimension reduction of the DCAE allowed us to improve the classification of the algorithm MiniRocket. Given the very small amount of data, our results are quite satisfying. In particular, the accuracy of the failure detection is 15% higher than the one obtained by the machine learning algorithm tested by SII. Moreover, the results could be substantially improved if the algorithm was trained on a larger dataset, which would be possible as the training data are simulated. The algorithms that we developed were trained and tested on artificial data produced by the flight simulator X-plane. However, the trained algorithms could be used to predict new unlabelled data, and in particular data from real flight tests. The advantage of our method is that we provide an accurate classification with small computational costs, as we extract the most important information among the data and use a fast classifier.

The classification of errors and flight scenarios will allow SII to compare the results obtained with the results of the research unit in order to generate new flight scenarios that may not have been previously considered. This will also allow the detection of the context of possible failures (type of failure and time when it occurs).

7 Access to the GitHub repository

The codes are available on the following GitHub repository : https://github.com/julie743/SII_comportement_vol_ST

References

[Alqahtani et al., 2021] Alqahtani, A., Ali, M., Xie, X., and Jones, M. (2021). Deep time-series clustering: A review. *Electronics*, 10:3001.

[Baydogan et al., 2013] Baydogan, M., Runger, G., and Tuv, E. (2013). A bag-of-features framework to classify time series. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35:2796–802.

[Cuturi, 2011] Cuturi, M. (2011). Fast global alignment kernels. In *International Conference on Machine Learning*.

[Dau et al., 2018] Dau, H. A., Bagnall, A. J., Kamgar, K., Yeh, C. M., Zhu, Y., Gharghabi, S., Ratanamahatana, C. A., and Keogh, E. J. (2018). The UCR time series archive. *CoRR*, abs/1810.07758.

[Dempster et al., 2019] Dempster, A., Petitjean, F., and Webb, G. I. (2019). ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *CoRR*, abs/1910.13051.

[Dempster et al., 2021] Dempster, A., Schmidt, D. F., and Webb, G. I. (2021). MiniRocket. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery &amp Data Mining*. ACM.

[Dempster et al., 2022] Dempster, A., Schmidt, D. F., and Webb, G. I. (commit 0b1c245 on 11 April 2022). Github repository: minirocket. https://github.com/angus924/minirocket. Online, accessed October 2022.

[Deng et al., 2013] Deng, H., Runger, G. C., Tuv, E., and Martyanov, V. (2013). A time series forest for classification and feature extraction. *CoRR*, abs/1302.2277.

[Faouzi, 2022a] Faouzi, J. (2022a). Time Series Classification: A review of Algorithms and Implementations. In Kotecha, K., editor, *Machine Learning (Emerging Trends and Applications)*. Proud Pen.

[Faouzi, 2022b] Faouzi, J. (2022b). Time Series Classification: A review of Algorithms and Implementations. In Kotecha, K., editor, *Machine Learning (Emerging Trends and Applications)*. Proud Pen.

[Fawaz et al., 2019] Fawaz, H. I., Lucas, B., Forestier, G., Pelletier, C., Schmidt, D. F., Weber, J., Webb, G. I., Idoumghar, L., Muller, P., and Petitjean, F. (2019). Inceptiontime: Finding alexnet for time series classification. *CoRR*, abs/1909.04939.

[Grabocka et al., 2014] Grabocka, J., Schilling, N., Wistuba, M., and Schmidt-Thieme, L. (2014). Learning time-series shapelets. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[Laurent, 2023] Laurent, B. (2022-2023). *Autoencoders and Generative Adversarial Netwoks - High Dimensional and Deep Learning*.

[Lines et al., 2012] Lines, J., Davis, L. M., Hills, J., and Bagnall, A. (2012). A shapelet transform for time series classification. In *Knowledge Discovery and Data Mining*.

[Lines et al., 2016] Lines, J., Taylor, S., and Bagnall, A. (2016). Hive-cote: The hierarchical vote collective of transformation-based ensembles for time series classification. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1041–1046.

[Lucas et al., 2018] Lucas, B., Shifaz, A., Pelletier, C., O'Neill, L., Zaidi, N. A., Goethals, B., Petitjean, F., and Webb, G. I. (2018). Proximity forest: An effective and scalable distance-based classifier for time series. *CoRR*, abs/1808.10594.

[Malcom and Ignacio, 2022] Malcom, M. and Ignacio, O. (commit be3c787 on November 2022). Github repository: timeseriesai/tsai/10_time_series_classification_and_regression_with_minirocket.ipynb. https://github.com/timeseriesAI/tsai/blob/main/tutorial_nbs/10_Time_Series_Classification_and_Regression_with_MiniRocket.ipynb. Online, accessed November 2022.

[Sakoe and Chiba, 1978] Sakoe, H. and Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49.

[Salmina et al., 2021] Salmina, L., Castello, R., Stoll, J., and Scartezzini, J.-L. (2021). Dimensionality reduction and clustering of time series for anomaly detection in a supermarket heating system. *Journal of Physics: Conference Series*, 2042(1):012027.

[Schäfer and Leser, 2017] Schäfer, P. and Leser, U. (2017). Fast and accurate time series classification with WEASEL. *CoRR*, abs/1701.07681.

[Schäfer, 2015] Schäfer, P. (2015). The boss is concerned with time series classification in the presence of noise. *Data Mining and Knowledge Discovery*, 29(6):1505–1530.

[Schäfer and Högqvist, 2012] Schäfer, P. and Högqvist, M. (2012). Sfa: A symbolic fourier approximation and index for similarity search in high dimensional datasets. pages 516 – 527.

[Senin and Malinchik, 2013] Senin, P. and Malinchik, S. (2013). Sax-vsm: Interpretable time series classification using sax and vector space model. In *2013 IEEE 13th International Conference on Data Mining*, pages 1175–1180.

[Shifaz et al., 2019] Shifaz, A., Pelletier, C., Petitjean, F., and Webb, G. I. (2019). TS-CHIEF: A scalable and accurate forest algorithm for time series classification. *CoRR*, abs/1906.10329.

[Sodagudi et al., 2022] Sodagudi, S., Manda, S., Smitha, B., Chaitanya, N., Ahmed, M. A., and Deb, N. (2022). Eeg signal processing by feature extraction and classification based on biomedical deep learning architecture with wireless communication. *Optik*, 270:170037.