UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Dipartimento di Scienze Pure e Applicate

Corso di Laurea Magistrale in Informatica e Innovazione Digitale

Master's thesis

MACHINE LEARNING TECHNIQUES FOR TIME SERIES ANALYSIS

Supervisor:
Prof. Valerio Freschi

Candidate:
Dott. Marzio Della Bosca

Academic Year 2024-2025

To Democritus, Giordano Bruno, and Alan Turing,
and to all whose questions give direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years time series analysis has achieved a very important role in many application contexts due to the large amount of data that can be recorded in short times and at very low cost. In many cases the analysis of time series, univariate and multivariate, form the basis of applications ranging from finance to medicine, as well as from engineering to the Internet of Things (IoT); at same time the development of increasingly powerful machine learning and high-performance neural network models has enabled new approaches to data modeling and analysis.

Work with time series present many challenges that make the analysis of this kind of data more articulated with respect to others. For instance, deciding whether or not dividing the series stream into windows for further analysis, e.g. for classification, is a design choice that severely impacts the performance of the downstream processes. This can be done through splitting the time series into fixed time intervals that will be used as input in the models' training operations.

This thesis tries to explore, and compare, several different machine learning techniques applied in classification tasks in different time series analysis contexts; in particular the work focuses on the comparison between tree models and neural network models (in particular by means of a model widely used in the field of time series classification). The rationale behind these comparison is given by the fact that at the state of the art "deep" models, usually, are known as the best option for classification in time series contexts.

This is generally accepted because even if neural networks are way more complex than the shallow ones they have the quality of having a greater capacity for generalization and are also designed to handle the input the way is recorded, or in other words, they are usually designed to perform inferences by directly acquiring the raw data, therefore they are simpler to use as they avoid the feature extraction phase, which is almost a must for less complex (and less deep) models. Anyway, this greater generalization capacity has a trade off: a greater architectural complexity implies an higher number of parameters, and because models training operations are, essentially, the seeking of the global minimum of the cost function, the increasing of the number of the weights involves in the increasing of the cost function dimension to optimize, and this implies a subsequently higher computational cost of training.

Following this introduction, Chapter 2 establishes the baseline by providing a comprehensive overview of time series classification with a specific focus on the algorithms employed in this study including decision trees, ensemble models such as Histogram-based Gradient Boosting (HGB) and deep learning architectures like the Fully Convolutional Network (FCN). Chapter 3 introduces the core methodological contribution of this work: the development of the FCN_{c22} model. This chapter details the integration of the Catch22 feature extraction algorithm as a secondary input stream, exploring a hybrid approach that combines raw temporal data with computed synthetic features to enhance predictive power. Chapter 4 presents the experimental evaluation through three distinct case studies; ranging from celestial body classification to human activity recognition covering both univariate and multivariate time series.

Within these case studies, a comparative analysis is performed between HGB, Decision Trees, the standard FCN, and the modified FCN_{c22} evaluating them based on accuracy and computational complexity, specifically training and inference times. The chapter also investigates performance trends across varying window sizes to ensure a coherent comparison of model scalability. Finally, Chapter 5 summarizes the findings and offers concluding remarks on the effectiveness of hybrid feature-based neural networks.

To summarize, tree-based models as Decision Trees and Hist Gradient Boosting Classifiers offer an alternative computational light weight solution with respect to Neural Network architectures, although neural networks remain a benchmark, in terms of accuracy, in different contexts and scenarios, such as time series analysis.

Chapter 2

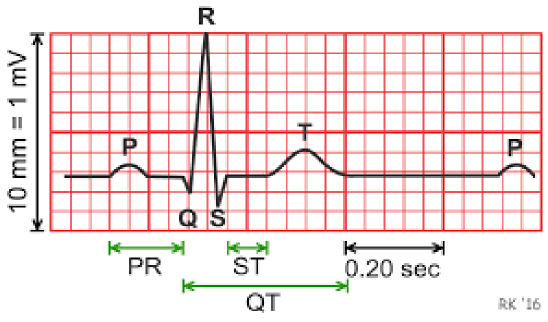
Baseline Models

The following chapter will introduce the kinds of model used in the thesis and the input' contexts by providing some basic background. The work defined in this thesis aims at comparing different machine learning architectures and different approaches in the time series analysis, in very different contexts, in order to gain knowledge regarding the design of efficient classifiers.

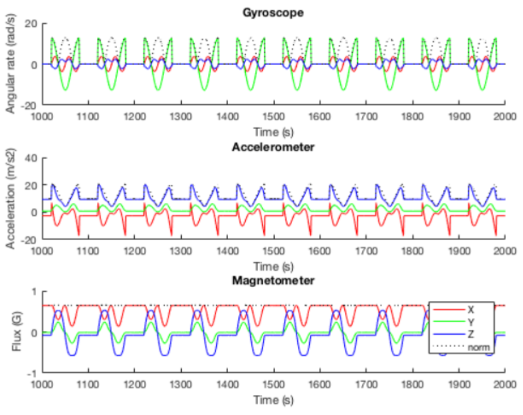
2.1 Time Series

A time series is a list of values which numerically represents a quantity underlying a phenomenon, sampled at fixed or variable intervals. A time series can be classified in univariate or multivariate: in the univariate time series case an only single time dependent variable is considered.

In Figure 2.1a, an example of univariate time series is reported, namely a portion of an electrocardiogram; Figure 2.1b refers to a multivariate time series representing IMU phone sensors' signals, where the IMU (Inertial Measurement Unit) combines an accelerometer for linear acceleration, a gyroscope for angular velocity, and a magnetometer for magnetic field to track orientation.



(a) Univariate: ECG



(b) Multivariate: Mobile IMU sensors

Figure 2.1: Examples of univariate and multivariate time series.

Mathematically, a multivariate time series can be viewed as a set described as:

$$X = \{\mathbf{x}_t \in \mathbb{R}^d \mid t = 1, 2, \ldots, T\} \tag{2.1}$$

Where d is the cardinality of the set of the recorded time-dependent variables', in the univariate cases d is equal to 1 hence we can say that univariate time series are particular cases of the multivariate ones. T is the number of recorded samples which have been included in the series. Our goal, in time series analysis, is to identify patterns, dependencies, and regularities within these sequences in order to perform predictions, classifications.

2.2 Classification Algorithms

In this thesis machine learning (ML) techniques will be used for classification algorithms. In short, ML techniques all use the same mathematically strategy: they minimize the "cost function" in order to get better predictions ([1]). For simplicity, assume that our task is a supervised binary classification task where we define the ground truth as $y_i = 0$ for the first class instances and $y_i = 1$ for the second class instances. Having defined these targets we can establish our cost function as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.2}$$

Where:

- y_i is the ground truth;
- $\hat{y}_i$ is the predicted probability.

The predicted probability $\hat{y}_i$ can be computed through various methods; generally, techniques are adopted to ensure the cost function is smooth, thereby facilitating the optimization algorithm in its search for the minimum. As with any closed function on an ordered set, such smoothness is crucial for convergence, a common example is through the logistic sigmoid function:

$$\hat{y}_i = \sigma(\mathring{\mathbf{X}}_i) = \frac{1}{1 + e^{-\mathring{\mathbf{X}}_i^T \mathbf{W}}} \tag{2.3}$$

where $\mathring{\mathbf{X}}_i$ represents the feature vector derived from the dataset and $\mathbf{W}$ is the weight vector which the algorithm try to learn through the training process. As with any closed function on an ordered set, this formulation allows us to find the minimum of the cost function through optimization.The defined equation in 2.2 shows specific loss function, which is the binary cross entropy (or BCE). In this framework $\mathring{\mathbf{X}}$ represents the set of constants (or feature vector) while $\mathbf{W}$ is the set of the independent variables; the feature vector $\mathring{\mathbf{X}}$ is augmented to include the model bias; this term defines the intercept of the decision boundary, effectively shifting the activation threshold. Assuming the **differentiability** of the cost function over the parameter space, the training process involves the continuous updating of the $\mathbf{W}$ vector: this is performed by the machine learning algorithm that seeks the minimum of $\mathcal{L}$ through the gradient descent algorithm:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \alpha d_{k-1} \tag{2.4}$$

Where:

- $\mathbf{W}_k$ is the weight vector at the k-th step of the algorithm;
- α is the learning rate;
- d_{k-1}, is obtained by $d_{k-1} = -\nabla_w \mathcal{L}(w_{k-1})$.

The term d_{k-1} denotes the steepest descent direction: representing the negative slope of the function at that specific point given the weight vector $\mathbf{W}$. This direction is determined by computing the gradient of the cost function with respect to the weights, $\nabla\mathcal{L}(\mathbf{w})$, which identifies the path of maximum local decrease to guide the optimization process.

While in many models this update follows the iterative step $w_k = w_{k-1} + \alpha d_{k-1}$, in **Decision Trees** the learned parameters $\mathbf{W}$ define the execution path of the algorithm; they dictate the logic that guides the input from the **root** to the specific **leaf** responsible for the final prediction.The objective remains to find the optimal configuration of $\mathbf{W}$ that minimizes the total error of the cost function.

2.2.1 Decision Trees

A decision classification tree model is a non-parametric supervised learning model used for classification tasks, decision trees are highly intuitive and powerful methods in machine learning; their primary strength lies in their explainability, they operate as "white box" models because the logic behind any given decision is easily traceable and explainable. In Figure 2.2 an example of a DT is given. The figure illustrates a Decision Tree classifier trained on the Iris dataset where each node represents a binary split based on a specific feature threshold aimed at isolating the three iris species: the decision process moves from the root to the leaves, where the *Gini impurity* index measures node homogeneity and the *value* array indicates the class distribution of samples at each step.

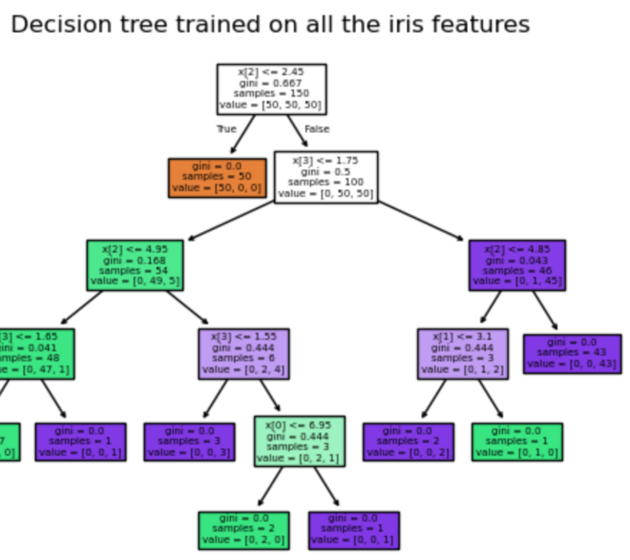

Decision tree trained on all the iris features
x[2] <= 2.45 gini = 0.667 samples = 150 value = [50, 50, 50]
True
False
gini = 0.0 samples = 50 value = [50, 0, 0]
x[3] <= 1.75 gini = 0.5 samples = 100 value = [0, 50, 50]
x[2] <= 4.95 gini = 0.168 samples = 54 value = [0, 49, 5]
x[2] <= 4.85 gini = 0.043 samples = 46 value = [0, 1, 45]
x[3] <= 1.65 gini = 0.041 samples = 48 value = [0, 47, 1]
x[3] <= 1.55 gini = 0.444 samples = 6 value = [0, 2, 4]
x[1] <= 3.1 gini = 0.444 samples = 3 value = [0, 1, 2]
gini = 0.0 samples = 43 value = [0, 0, 43]
gini = 0.0 samples = 47 value = [0, 47, 0]
gini = 0.0 samples = 1 value = [0, 0, 1]
gini = 0.0 samples = 3 value = [0, 0, 3]
x[0] <= 6.95 gini = 0.444 samples = 3 value = [0, 2, 1]
gini = 0.0 samples = 2 value = [0, 0, 2]
gini = 0.0 samples = 1 value = [0, 1, 0]
gini = 0.0 samples = 2 value = [0, 2, 0]
gini = 0.0 samples = 1 value = [0, 0, 1]

Figure 2.2: Decision Tree Prediction Workflow example.

Stakeholders can easily visualize the decision path, moving from the root node through branches (splits based on feature values) to a leaf node (the final prediction) allowing for immediate verification and clear explanation of the model's behavior. Decision tree architectures present several problematic aspects: first of all they can can create over-complex trees that do not generalize well over training (this is a kind of overfitting), even there exist methods (e.g. pruning) in able to mitigate this behavior. Decision trees can be unstable because this class of algorithms is highly sensitive to small variations in the training data, often leading to the generation of entirely different tree structures as remarked by SciKit library [2]; in general, their predictions, tend to be heavily biased toward the training set, resulting in poor generalization performance on unseen data.

2.2.2 Ensemble Models

Ensemble machine learning methods combine the predictions of several base estimators. Figure 2.3 illustrates the concept of Ensemble Learning, specifically an "Ensemble of Trees" (e.g. Random Forest). This method combines the predictions of multiple base estimators to improve robustness and generalizability compared to a single model, usually, the final outcome is determined through a voting mechanism: which aggregates the individual results to reduce variance and prevent overfitting. This kind of models can be applied to any base learner, in averaging methods such as Bagging methods, model stacking, Voting, etc as remarked in Scikit library [3].

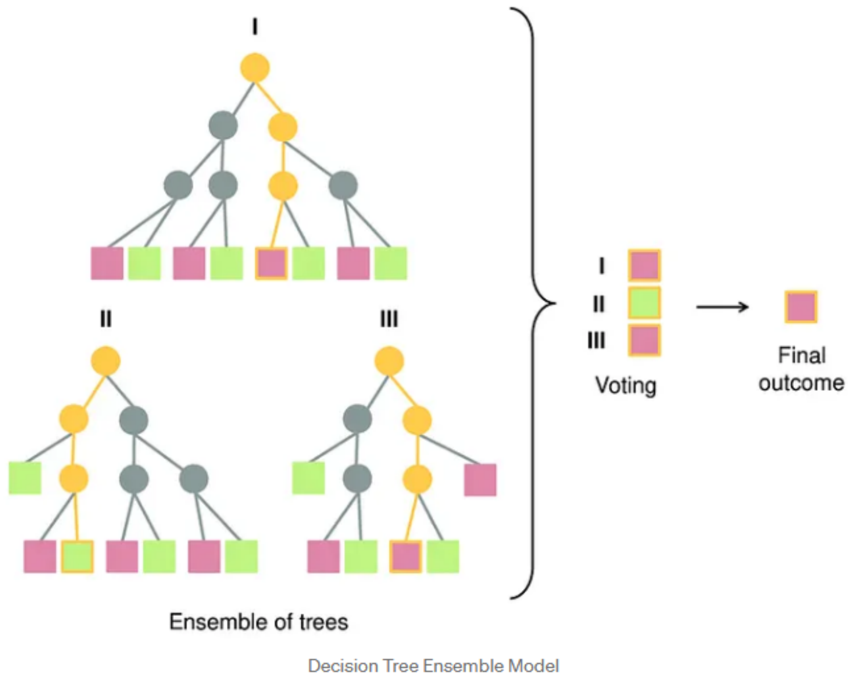


Figure 2.3: Ensemble Tree Prediction Workflow with voting example.

The two most used class of ensemble methods are the Gradient boosted tree algorithms and the Random Forest ones. The Random Forest algorithm is a meta estimator and uses many single treed algorithms to make predictions as remarked by Scikit library [4]. To mitigate the overfitting problem each individual tree is trained on a different bootstrap sample of the dataset; the final prediction is then obtained by averaging the outputs of all trees in the forest rather than relying on a single model. Gradient boosted tree algorithms are left for the next section.

Histogram Gradient Boosting Classification Tree

As said in the last section ensemble methods use multiple tree based algorithms (e.g. decision tree) in order to achieve better predictive performance compared to a single tree-based approach. In this thesis Histogram Gradient Boosting classifier is used as reference model for comparison given its flexibility (i.e. its capability of achieving particularly good performances on many types of input). This algorithm is an optimization of the Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) as remarked by Scikit library [5]. The Gradient Boosting algorithm is a meta estimator which combines an ensemble of weak learners (shallow decision trees). These models are added sequentially: each new tree is trained to minimize the residual errors of the previous ones by applying gradient descent to a cost function. Figure 2.4 illustrates the workflow of a Histogram Gradient Boosting Classifier, which accelerates training by binning continuous feature values into discrete bins. Specifically, it illustrates the architecture of the THGB model, where the Histogram Gradient Boosting (HGB) component plays a critical role in identifying high quality Ligand Receptor Interactions (LRIs).

In the work proposed by Zhou et al. [6], the Histogram technique is specifically used during the optimization of the decision tree nodes. Instead of searching through every individual feature the model bins continuous biological features into discrete intervals: this discretization allows the algorithm to construct histograms of gradients which are used to efficiently determine the optimal split points. By focusing on these bins rather than raw data points, THGB significantly accelerates the selection of representative features and the search for the global minimum of the cost function. The process involves:

1. Feature binning;
2. Parallel histogram calculation across multiple workers;
3. Identifying the optimal split point based on these histograms;
4. Performing the node split.

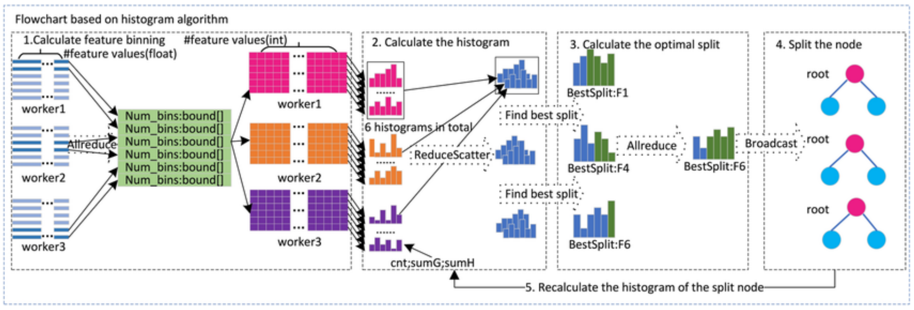


Figure 2.4: Histogram Gradient Boosting flowchart example.

Histogram Gradient Boosting Classifier is an optimization of Gradient Boosting Classifier because the "Hist" has been built up to handle very large datasets, improving computational performances and memory management as remarked by Scikit library [5]. The main idea consists of quantizing the continuous features into a limited number of discrete bins (i.e. histograms); during the training phases each feature is subject to a discretization process up to a maximum number of bins, this process allows for faster identification of optimal split points, thus reducing the computational cost. Another important feature of this model is its capability of missing values handling: for each node the algorithm evaluates the expected gain by trial assigning samples with missing values to both the left and right branches, then selects the default direction that best minimizes the cost function estimating the most probable path based on the patterns identified during the training phase.

Among the model's main parameters we may list the following:

- `learning_rate`: manages the learning speed and the model shrinkage;
- `max_iter`: it defines the maximum number of boosting iterations (i.e. the number of trees built);
- `max_leaf_nodes`: it sets the maximum number of leaves per tree;
- `min_samples_leaf`: it enforces a minimum number of samples per leaf to reduce overfitting;
- `l2_regularization`: it adds an L2 regularization term
- `early_stopping`: it allows for premature termination of training if the model no longer improves on a validation set.

2.2.3 FCN Classifier

The Fully Convolutional Network (FCN) is a kind of class of fully connected Neural Network, which are based on for the strog use of Convolutional layers. The primary difference between a traditional machine learning algorithm and a Neural Network lies in the architectural complexity: a Neural Network is composed of multiple layers, each containing several computational units known as neurons:

- **Neurons:** Each unit performs a weighted sum of its inputs followed by a non-linear activation function: this allows the network to model complex relationships that a simple linear model cannot capture;
- **Layers:** High-level structures that organize units into functional groups, they are organized into an *Input Layer*, one or more *Hidden Layers*, and an *Output Layer*.

While a standard model might find a direct mapping from features to labels, a Neural Network uses its layered structure to perform hierarchical feature extraction, where each subsequent layer refines the information processed by the previous one. Neural networks are categorized as either Deep or non deep; depending on the number of hidden layers within the architecture.

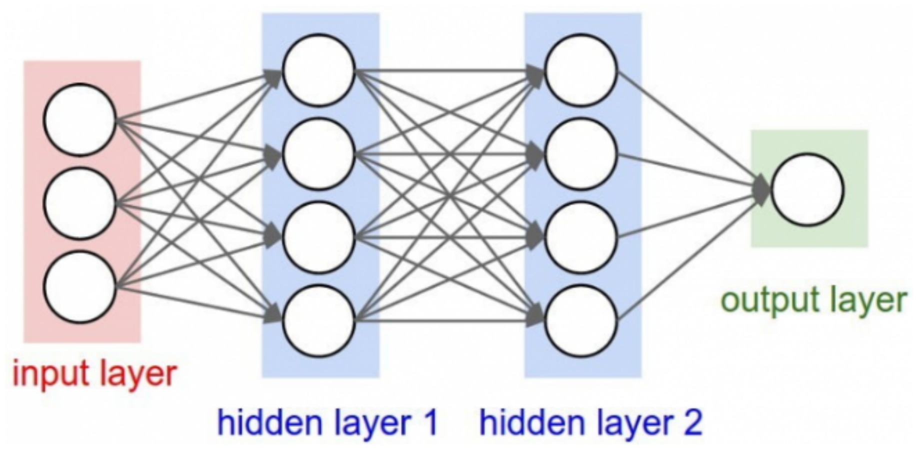


Figure 2.5: Deep Neural Network Architecture example.

Convolutional Network or CCNs are composed of multiple layers including:

- **Convolutional layers**: they contain the sets of learnable filters, based on the convolution operator;
- **Pooling layers**: they reduce the number of parameters and computations by performing a down-sampling of the representation from the filters;
- **Fully connected layers**: in this layers neurons have full connections to all activations in the previous layer.

The width and height of the filters contained in the Convolutional layers are smaller than those of the input volume: the filter slides across the input and the dot products between the input and filter are computed at every spatial position. Convolution is a mathematical operation commonly used in signal processing; it involves combining two functions (f and g) to produce a third function (i.e. $(f \circ g)(x) = f(g(x))$). Convolution in discrete terms can be represented as follow:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n-m], \quad \text{for } -\infty < m, n < \infty \tag{2.5}$$

The main purpose of the Pooling layers is to aggregate information in order to reduce the spatial resolution (lowering the parameters' number) mitigating the sensitivity of convolutional layers to the exact location of features, achieving approximate translation invariance ([7]).

The FCN used in this thesis is based on the baseline model proposed in by Wang et al. [8]; this architecture has been optimized for time series analysis and serves as a benchmark for these kinds of applications, the implementation used is an adaptation by Aeon, a scikit-learn compatible toolkit for time series machine learning tasks such as classification, from the GitHub implementation by Fawaz et al. [9]; which is a result of the paper "Deep learning for time series classification: a review" also redacted by Fawaz [10]. This latter paper is a comprehensive review, extending the work presented in the first mentioned publication (i.e. [8]), this architecture defines the network as a sequence of convolutional blocks ending with a global pooling operation. The FCN model is built from three 1D convolutional blocks followed by final classification layers. The FCN architecture used is shown in the next Table 2.1. Conv1D layers use `padding='same'` to force the output

Table 2.1: FCN Architectural Summary.

Layer Type	**Parameters**	**Input Shape**	**Output Shape**
`InputLayer`	–	(Time Steps, D)	(Time Steps, D)
Block 1:	Kernel Size: 8, Padding: same	(Time Steps, D)	(Time Steps, 128)
Block 2:	Kernel Size: 5, Padding: same	(Time Steps, 128)	(Time Steps, 256)
Block 3:	Kernel Size: 3, Padding: same	(Time Steps, 256)	(Time Steps, 128)
`GlobalAvgPool1D`	–	(Time Steps, 128)	(128)
`Dense`	Units: N, Activation: Softmax	(128)	(N)

sequence length (Time Steps) consistency through the blocks. Each of the three convolutional blocks follows the same structure applying a sequence of operations to the input time series where N represents the number of target classes (`nb_classes`). The sequential flow of operations can be seen as:

$$\begin{gathered}\text{Conv1D (128 Filters, } K=8) \rightarrow \text{BN} \rightarrow \text{ReLU}\\ \text{Conv1D (256 Filters, } K=5) \rightarrow \text{BN} \rightarrow \text{ReLU}\\ \text{Conv1D (128 Filters, } K=3) \rightarrow \text{BN} \rightarrow \text{ReLU}\\ \text{GlobalAveragingPooling}\\ \text{Dense (128, nb\_classes)} \rightarrow \text{SoftMax}\end{gathered}$$

In summary each block is made of:

1. **Conv1D Layer:** Applies 1D convolution across the time dimension.
2. **Batch Normalization (BN)** Normalizes the output of the previous layer, stabilizing and speeding up the training process.
3. **ReLU Activation:** Introduces non-linearity into the network.

The **Batch Normalization** layer is set to stabilize the learning process by normalizing the inputs of each layer: when training, the layer transforms the input x using the mean (μ_{batch}) and variance (σ^2_{batch}) of the current batch:

$$\hat{x} = \frac{x - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \epsilon}}, \quad y = \gamma \hat{x} + \beta \tag{2.6}$$

where γ and β are learnable parameters that allow the network to scale and shift the normalized values. The parameters γ and β in Equation 2.6 are learnable variables that allow the network to perform a transformation on the normalized input $\hat{x}$. Specifically, γ acts as a scaling factor that multiplies the normalized value, enabling the network to recover the original amplitude if γ matches the standard deviation σ. In parallel β serves as a shifting parameter that is added to the value, allowing the network to restore the original data position if β corresponds to the original mean μ. This mechanism ensures that the Batch Normalization layer can represent the identity transform, preserving the model's expressive power during training. During inference, the layer switches its behavior: instead of batch statistics, it uses moving averages of the mean and variance accumulated during the training phase as shown in Figure 2.6.

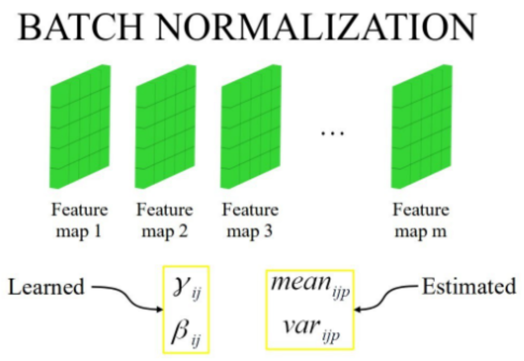


Figure 2.6: Batch Normalization algorithm.

This ensures that the normalization remains consistent even when processing a single sample at a time.

The **ReLU (Rectified Linear Unit)** is the activation function applied to the output of the convolutional and dense layers. It is defined as:

$$f(x) = \max(0, x) \tag{2.7}$$

This non linear transformation introduces sparsity in the network, as neurons with negative inputs are effectively deactivated and it mitigates the vanishing gradient problem: because as the derivative of the function is constant (1) for all positive inputs there is a faster and more stable convergence during the training of deep architectures.

Chapter 3

Machine learning models with Catch22 features

3.1 Catch22

The 22 Canonical Time-series Characteristics (or Catch22) are 22 characteristics selected through highly comparative time series analysis on several context applications by Lubba et al. [11]. In this publication the authors proposed a method to derive small sets of time series features that exhibit strong classification performance across a wide collection of time series problems, the resulting collected features are also choosed to be less redundant as possible.

The proposed, and used, pipeline selects a reduced set of high performing features while minimizing inter feature redundancy; at the beginning a statistical prefiltering is performed to identify and exclude features whose performance is indistinguishable from random noise, to achieve this the authors derived null accuracy distributions for each feature by repeatedly classifying shuffled class labels. Features whose p-values, combined across datasets, indicated performance consistent with random number generators were discarded. The resulting set of 22 significant features that have been proposed are listed in the following.

- **Distribution** selected features:
  - `DN_HistogramMode_5`: Mode of z-scored distribution (5-bin histogram).
  - `DN_HistogramMode_10`: Mode of z-scored distribution (10-bin histogram).

- **Simple temporal statistics** selected features:
  - `SB_BinaryStats_mean_longstretch1`: Longest period of consecutive values above the mean.
  - `DN_OutlierInclude_p_001_mdrmd`: Time intervals between successive extreme events above the mean.

  - `DN_OutlierInclude_n_001_mdrmd`: Time intervals between successive extreme events below the mean.

- **Linear autocorrelation** selected features:
  - `CO_f1ecac`: First $1/e$ crossing of autocorrelation function.
  - `CO_FirstMin_ac`: First minimum of autocorrelation function.
  - `SP_Summaries_welch_rect_area_5_1`: Total power in lowest fifth of frequencies in the Fourier power spectrum.
  - `SP_Summaries_welch_rect_centroid`: Centroid of the Fourier power spectrum.
  - `FC_LocalSimple_mean3_stderr`: Mean error from a rolling 3-sample mean forecasting.

- **Nonlinear autocorrelation** selected features:
  - `CO_trev_1_num`: Time-reversibility statistic, $\langle (x_{t+1} - x_t)^3 \rangle$.
  - `CO_HistogramAMI_even_2_5`: Automutual information, $m = 2, \tau = 5$.
  - `IN_AutoMutualInfoStats_40_gaussian_fmmi`: First minimum of the automutual information function.

- **Successive differences** selected features:
  - `MD_hrv_classic_pnn40`: Proportion of successive differences exceeding 0.04σ.
  - `SB_BinaryStats_diff_longstretch0`: Longest period of successive incremental decreases.
  - `SB_MotifThree_quantile_hh`: Shannon entropy of two successive letters in equiprobable 3-letter symbolization.
  - `FC_LocalSimple_mean1_tauresrat`: Change in correlation length after iterative differencing.
  - `CO_Embed2_Dist_tau_d_expfit_meandiff`: Exponential fit to successive distances in 2-d embedding space.

- **Fluctuation Analysis** selected features:
  - `SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1`: Proportion of slower timescale fluctuations that scale with DFA (50% sampling).
  - `SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1`: Proportion of slower timescale fluctuations that scale with linearly rescaled range fits.

- Other selected features:
  - `SB_TransitionMatrix_3ac_sumdiagcov`: Trace of covariance of transition matrix between symbols in 3-letter alphabet.
  - `PD_PeriodicityWang_th0_01`: Periodicity measure.

The choice of Catch22 is justified by its ability to represent time series through a small set of interpretable characteristics, effectively overcoming the computational overhead typical of larger libraries like TSFEL. Furthermore as discussed in the cited pubblication by Lubba et al. [11] this feature-based approach is particularly suited for datasets where classes do not have a characteristic which are temporally aligned shape but have characteristic dynamical properties.

3.2 FCN$_{c22}$ Classifier

To enhance the capabilities of the standard Fully Convolutional Network (FCN) classifier (as derived from the `aeon` library baseline by Fawaz et al. [9]), a significant architectural modification was introduced to handle a supplementary input data stream.

The modified architecture incorporates two distinct feature paths before the final classification layer: the core structure of the FCN for time series feature extraction remains unchanged processing the primary time series input (X_{ts}) through three identical convolutional blocks (Conv1D $\rightarrow$ BatchNormalization $\rightarrow$ ReLU). After the convolutional blocks the time dimension is collapsed using `GlobalAveragePooling1D` (GAP). This operation yields a feature vector (F_{ts}) of a fixed size, concurrently an offline feature extraction is performed using the Catch22 algorithm on the same input time series obtaining an auxiliary feature array (X_{C22}) with a fixed dimension of ($d \times 22$) where d is the number of dimensions (or channels) in the input time series (X_{ts}). This second vector (X_{C22}) is then concatenated with the FCN's GAP layer output (F_{ts}), the resulting merged vector (F_{merged}) is used as input for the final dense classification layer.

$$F_{merged} = F_{ts} \oplus X_{C22}$$

The complete architecture, showing the two parallel input streams and the fusion step, is detailed in Table 3.1.
The FCN architecture proposed by Wang et al. [9] was originally conceived as a "pure end-to-end" baseline: because this architecture has beed developed to avoid any manual feature crafting. Integrating synthetic descriptors from Catch22 provides a complementary informational layer that raw convolutions might overlook, especially in domain specific tasks. To validate the effectiveness of these combined features the Histogram-based Gradient Boosting Classifier (HGB) has been selected as a benchmark. While the FCN is designed to learn representations directly from temporal signals, HGB is specifically optimized for tabular data, such as the output of the Catch22 algorithm. A standard Decision Tree (DT) has been also included as a fundamental benchmark because the Decision Tree provides a

Table 3.1: FCN$_{c22}$ Architectural Summary.

Layer Type	**Function / Source**	**Input Shape**	**Output Shape**
`Input TS`	Time Series Input	(Time Steps, D)	(Time Steps, D)
`FCN Blocks (1-3)`	Conv1D $\rightarrow$ BN $\rightarrow$ ReLU	(Time Steps, D)	(Time Steps, 128)
`GlobalAvgPool1D`	Generates F_{ts}	(Time Steps, 128)	(128)
`Input C22`	Catch22 Features (X_{C22})	–	$(d \times 22)$
`Concatenate`	Merges F_{ts} and X_{C22}	(128) and $(d \times 22)$	$(128 + d \times 22)$
`Dense`	Units: N, Activation: Softmax	$(128 + d \times 22)$	(N)

baseline for the inherent discriminative power of the Catch22 features through a simple, hierarchical decision process. This setup allows us to establish a performance baseline for an exclusively feature based approach, enabling a rigorous comparison between:

This expands our rigorous comparison to four distinct approaches:

- **FCN**: specialized in processing raw signals [9];
- **Decision Tree (DT)**: baseline for interpretability;
- **HGB**: optimized for maximizing performance on the same tabular statistical features;
- **FCN$_{C22}$**: integrates both raw temporal patterns and synthetic features.

The inclusion of the Decision Tree architecture as a benchmark model allows us to assess whether the complexity of the hybrid deep learning model is justified compared to a simpler architecture. This comparison aims also to verify whether fusing the "end-to-end" representation learning of deep networks with the targeted statistical insights of Catch22 yields a superior classification synergy, as opposed to relying on either the raw signal or the handcrafted features in isolation.

Chapter 4

Case studies

In this chapter we will describe specific case studies that have been taken as reference for evaluating the performance of the model architectures previously introduced, in particular, we applied the proposed models and techniques to two univariate time series classification case studies and, finally, to a multivariate one. As explained in the previous chapters, the reference architectures are:

- Decision trees;
- Ensemble trees (HGB);
- Fully Convolutional Neural Network.

The aim of the explorations is to compare very different architectures both when they take the "raw" time series as input and when they take synthetic data resulting from extraction of the Aeon project Catch22 algorithm: except for the modified fully convolutional network with dual input, all models were evaluated using both raw windowed time series and Catch22 feature representations as input. The datasets of the univariate cases were taken from the repository of temporal series made available by AEON in its repository [12]; while for the exploration of the multivariate case it was chosen to operate in the Human Activity Recognition (HAR) context using the Motion Sense dataset made available by Malekzadeh et al. [13]. All experiments were conducted using CPU resources only: the system was equipped with an AMD Ryzen 7 5700 processor with 16 logical cores running at approximately 3.7 GHz and 32 GB of RAM.

4.1 Case Study 1: Celestial Body Classification from Spectrum

The first case study of univariate time series applications is the "Kepler Light Curves" dataset, made available by the AEON time series repository. This dataset has been used in "Classifying Kepler light curves for 12000 A and F stars using supervised feature-based machine learning" by Barbara, Nicholas H and Bedding et al. [14] to develop a machine learning pipeline for classifying Kepler light curves for specific stars. In this dataset each case is a light curve (brightness of an object sampled over time) from NASA's Kepler mission: they're data are obtained from 3 month long measurements with a sample rate of 30 min where missing values has been handled by interpolation techniques. In the dataset made by Nicholas H Barbar et al. [14] there are 7 different kind of objects, which represent the 7 classes of the recognition problem:

1. **Contact eclipsing binary** stars are binary systems in which the two stellar components are in contact or share a common envelope, these stars periodically eclipse each other when observed edge-on from **Earth**.
2. **Detached eclipsing binary** stars are binary systems in which both stellar components are well within their respective Roche lobes, due to this, these stars retain an approximately spherical shape and do not experience significant surface deformation.
3. **Delta Scuti variable** stars are pulsating stars characterized by A or F-type spectrum. They are sometimes referred to as dwarf Cepheids when the amplitude of variability in the V band exceeds 0.3 magnitudes.
4. **Gamma Doradus variable** stars exhibit luminosity variations caused by non-radial surface pulsations: they are typically young late A or early F-type main-sequence stars with brightness variations of about 0.1 magnitudes and periods on the order of one day.
5. **Non-variable** stars are celestial objects whose brightness remains approximately constant over time.
6. **Rotational variable** stars show brightness variations due to stellar rotation. These variations are caused by surface inhomogeneities such as starspots (same as sunspot) or by geometric distortions such as ellipsoidal shapes in close binary systems.
7. **RR Lyrae variable** stars are periodic variable stars commonly found in globular clusters; they are widely used as "standard candles" for measuring extragalactic distances and play a key role in the cosmic distance ladder.

4.1.1 Dataset and Pre-processing

At the beginning, the whole dataset, was windowed using a specific function. In the context of time series, the windowing process is fundamental as it allows to define a time period to use for analyzing similarities in the series. In the context of classification, choosing the window size in the windowing process involves choosing the set of information on which the machine learning algorithm should base its classification. The dataset features have been acquired at a sampling rate of 5.5×10^{-4} Hz, which is equivalent to one sample every 30 minutes spanning a total duration of three months. To evaluate the performance of the different architectures the models were trained using input windows corresponding to 20%, 35%, 50%, and 100% of the total signal length, respectively. This function windowizes the dataset, taking the un windowed dataset and the label vector as input, it execute the windowing process and return the windowed dataset and the new label vector, consistent with the windowing operation performed. The pseudocode of which is given below:

Algorithm 1 Windowing of time series data

```
Require: Dataset $\mathbf{X}$, labels $\mathbf{y}$, window size $w$, flag $need\_remain$
Ensure: Windowed samples $\mathbf{X}_w$ and corresponding labels $\mathbf{y}_w$
 1: Initialize empty lists $\mathbf{X}_w \leftarrow \emptyset$, $\mathbf{y}_w \leftarrow \emptyset$
 2: for each sample index $i$ in the dataset do
 3:     Extract time series $\mathbf{s} \leftarrow \mathbf{X}[i]$
 4:     Extract label $l \leftarrow \mathbf{y}[i]$
 5:     Compute number of complete windows $n \leftarrow \lfloor |\mathbf{s}|/w \rfloor$
 6:     for $j = 0$ to $n - 1$ do
 7:         $start \leftarrow j \cdot w$
 8:         Extract window $\mathbf{s}[start : start + w]$
 9:         Append window to $\mathbf{X}_w$
10:         Append label $l$ to $\mathbf{y}_w$
11:     end for
12:     if $need\_remain$ is true then
13:         Compute remaining samples $r \leftarrow |\mathbf{s}| \bmod w$
14:         if $r > 0$ then
15:             Extract last window of size $w$ with overlap
16:             Append window to $\mathbf{X}_w$
17:             Append label $l$ to $\mathbf{y}_w$
18:         end if
19:     end if
20: end for
21: Convert $\mathbf{X}_w$ and $\mathbf{y}_w$ to arrays return $\mathbf{X}_w$, $\mathbf{y}_w$
```

Because of the corresponding window lengths of 20%, 35%, 50%, and 100% to ensure a fair comparison across different windowing strategies we addressed the training set size by normalizing the number of examples. Models trained on windows of size d were configured to utilize half the number of examples compared to those trained on windows of size $2d$. We avoided systematic overlapping: a specific overlap was introduced exclusively between the final and penultimate windows of each time series; this adjustment was necessary to prevent data loss in cases where the chosen window size was not a perfect divisor of the total signal length ensuring that even the terminal segments of the series were included in the training process.

This consistency in the experimental setup allowed us to evaluate a critical trade off in time series analysis: whether the architectures perform better when provided with higher-dimensional context (i.e. larger window sizes) at the expense of the total number of training samples, or conversely, when trained on a larger volume of simpler, lower-dimensional examples.

Once the windowed dataset is obtained it is important to prepare it in order to make it usable by a ML algorithm, to best train the architectures. First, a normalization was performed to standardize the scale of the different variables and prevent features with larger orders of magnitude from disproportionately influencing the model training. Data values normalization, or scaling, was performed using the StandardScaler function, also from the scikit-learn library. This method transforms each element x_i according to Equation 4.1 :

$$z_i = \frac{x_i - \mu}{\sigma} \tag{4.1}$$

where:

- x_i is the original value of the variable;
- μ is the mean of the variable's values;
- σ is the standard deviation of the variable's values;
- z_i is the standardized value (z-score).

In order to have a method to verify the goodness of the model, when trained, it was used a split function, provided by the scikit library: this function performs a randomized split of the dataset based on a specified seed to ensure reproducibility, then it returns the partitioned training and testing sets along with their respective label (target) vectors., with an 80-20% division for training set - test set.

Another key point of the data preparation phase was the balancing phase in order to obtain not only more consistent comparisons but also classifications less dependent on the captured data, which could be a non representative distribution of the group of objects under examination. In this case we have an extremely unbalanced dataset:

- Contact eclipsing binary are the 12.96% of all the instances;
- Detached eclipsing binary are the 6.29% of all the instances;

- Delta Scuti variable are the 31.16% of all the instances;
- Gamma Doradus variable are the 19.86% of all the instances;
- Non-variable are the 15.24% of all the instances;
- Rotational variable are the 12.59% of all the instances;
- RR Lyrae variable are the 1.90% of all the instances.

For this reason a balancing operation was necessary to achieve a less case biased exploration. "Subsampling" was chosen as the technique to avoid producing synthetic examples in the training phase. The choosed algorithm is from imblearn.under_sampling python library [15] which under-sample the majority class(es) by randomly picking samples with or without replacement. In Algorithm 2 the pseudo-code of how datas are handled before the training:

Algorithm 2 Scaling, splitting and balancing of the dataset

```
Require: Dataset $\mathbf{X}$, labels $\mathbf{y}$, random seed $s$
Ensure: Scaled and balanced training set, scaled test set
 1: Initialize random undersampler with seed $s$
 2: Apply random undersampling on $\mathbf{X}$
 3: Apply random undersampling on $\mathbf{y}$
 4: Split $\mathbf{X}, \mathbf{y}$ into training and test sets using stratified sampling
 5: Extract Catch22 features from training set
 6: Extract Catch22 features from test set
 7: Scale training features using fitted scaler on raw data
 8: Scale test features using the same scaler
 9: Scale Catch22 training features using fitted scaler on catch22 features' data
10: Scale Catch22 test features using the same scaler
11: Convert labels to integer format
        return Scaled training set, scaled test set, scaled C22 Processed training, scaled
    C22 Processed test sets, training labels, test labels.
```

4.1.2 Experimental Results

In this subsection the results of the experimental evaluation conducted on the Kepler light curves dataset are reported: the study investigates the impact of window size selection and feature representation on model performance, comparing tree-based methods and neural networks in terms of accuracy as well as training and inference time.

The training of the Decision Tree was optimized using a `GridSearchCV` approach. This allowed for the systematic evaluation of different hyperparameter configurations to find the best trade-off between model complexity and predictive accuracy. Specifically, we tuned:

- `max_depth`: Controls the maximum height of the tree to prevent overfitting.
- `ccp_alpha`: Implements Cost Complexity Pruning, a technique used to prune insignificant branches and improve generalization.

The parameter grid used for the optimization was:

```
param_grid = {
    'max_depth': [3, 8, None];
    'ccp_alpha': [0.01, 0.05, 0.1].
}
```

The `HistGradientBoostingClassifier` was optimized through a grid search to refine the ensemble's learning behavior, the following hyperparameters were evaluated:

- `max_depth`: Limits the maximum depth of each individual tree in the boosting ensemble, controlling the complexity of the base learners.
- `learning_rate`: Governs the contribution of each tree to the final prediction. It scales the step size during the gradient descent process to balance training speed and model stability.

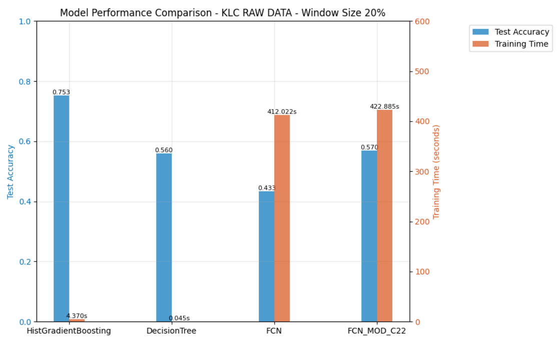
The configuration grid for the search was defined as follows:

```
param_grid = {
    'max_depth': [3, 8, None];
    'learning_rate': [0.1, 0.2, 0.3].
}
```

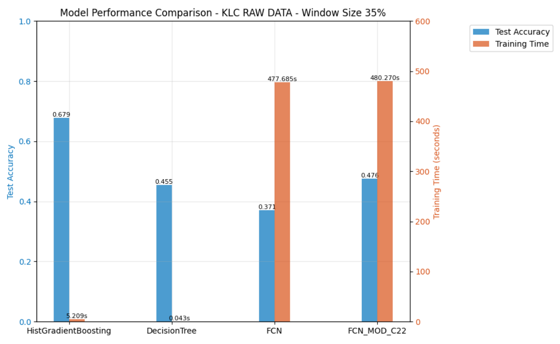
The following figures present a preliminary comparison of the proposed architectures; they show the classification accuracy and training time obtained by each model for different window sizes.

KLC Dataset: Raw Input Results

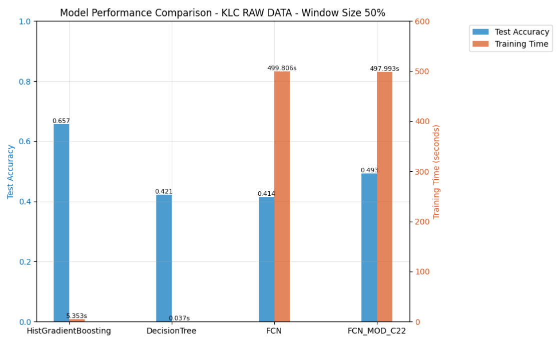
Figures 4.1 report the average Test accuracies and the average Training times for the architectures (Decision Trees, Hist Gradient Boosting Classifiers, Fully Connected Networks and modified FCNs) for the different windows sizes where the expected input is the raw time series of the signal. It is important to clarify that the training time reported for HGBs and DTs are the result of dividing the total time spent in training by the number of fits performed by the grid search function (45 fits). HGB consistently shows the best average accuracies, while DT is the fastest architecture. However, the difference of a few seconds between DT and HGB isn't enough to favor it over HGB, as HGB averages 20 points more accuracy for almost all window sizes. As expected in the case of raw time series inputs, an increase in the window size corresponds to a significant increase in training times. A small difference in accuracy trends can be observed between FCNs and trees. All architectures performed worst when the window size was set to 100% of the time series, however, trees



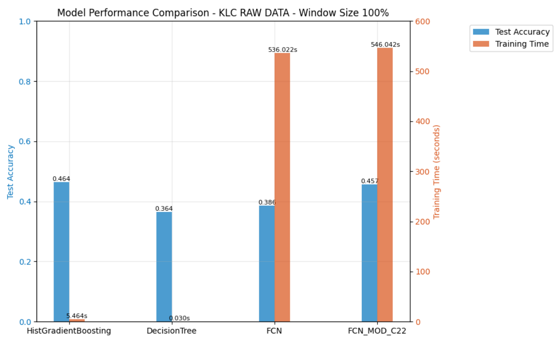
(a) Window size: 20%



(b) Window size: 35%



(c) Window size: 50%



(d) Window size: 100%

Figure 4.1: KLC performances using raw time series input.

consistently decreased in accuracy, while FCNs performed better at windows set to 50% versus 35%. The modified FCNs performed better than the FCNs in all the tests, a sign that the model is able to be positively influenced by the second input Catch22.

Table 4.1: Performance comparison of the models on KLC dataset using raw time series input.

Model	**Window Size**	**Accuracy**	**Training Time (s)**	**Prediction Time (s)**
HGB	20	0.7529	4.370	4.34×10^{-5}
DT	20	0.5600	0.045	2.87×10^{-6}
FCN	20	0.4329	412.022	2.69×10^{-3}
FCN_{c22}	20	0.5700	422.885	2.94×10^{-3}
HGB	35	0.6786	5.209	4.56×10^{-5}
DT	35	0.4548	0.043	—
FCN	35	0.3714	477.685	4.63×10^{-3}
FCN_{c22}	35	0.4762	480.270	4.89×10^{-3}
HGB	50	0.6571	5.353	1.21×10^{-4}
DT	50	0.4214	0.037	—
FCN	50	0.4143	499.806	7.96×10^{-3}
FCN_{c22}	50	0.4929	497.993	7.69×10^{-3}
HGB	100	0.4643	5.464	2.17×10^{-4}
DT	100	0.3643	0.030	—
FCN	100	0.3857	536.022	1.39×10^{-2}
FCN_{c22}	100	0.4571	546.042	1.39×10^{-2}

Table 4.1 reports the overall average results obtained using raw time series as input alongside the corresponding average inference times. For the Decision Tree (DT) models the inference times are frequently marked as "—" because their values are too small to be reliably measured by the Python interpreter.

The average inference time for each model was computed during the testing phase by dividing the total prediction time on the test set by the number of test samples, consequently, the inference time for Decision Trees is so minimal that, when subdivided, it falls below the precision limits of Python's timing utilities, making it impossible to represent. Interestingly, the longest average inference time for the DT model occurs with the smallest window size.

This counter intuitive result is likely due to the fact that when dealing with extremely efficient models the actual computation time for a single prediction becomes comparable to, or even eclipsed by, the software overhead. In this regime the time required by the Python interpreter or the Jupyter kernel to manage data structures and feed the input to the model can become an important factor.

KLC Dataset: Catch22-Based input Results

Here are shown the results using the Catch22 features. In table 4.2 are shown the Average Catch22 feature extraction time per sample for different window sizes.

Table 4.2: Average Catch22 extraction time per sample for different window sizes on KLC dataset.

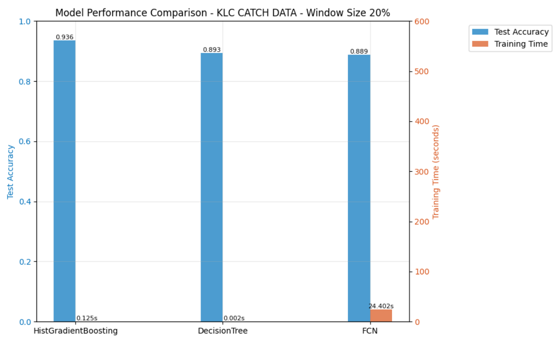
Window Size (% of signal)	Tot. Ext. Time [s]	Ext. per Sample [s]
20	9.04	0.00137
35	9.35	0.00236
50	10.44	0.00396
100	11.34	0.00860

As shown in the table, the extraction of the 22 synthetic features results in the expected increase in computational cost, as reflected by the higher processing times required for the catch22 feature extraction per sample. However, the result was not a given since increasing the window size leads to a reduction in the number of examples.

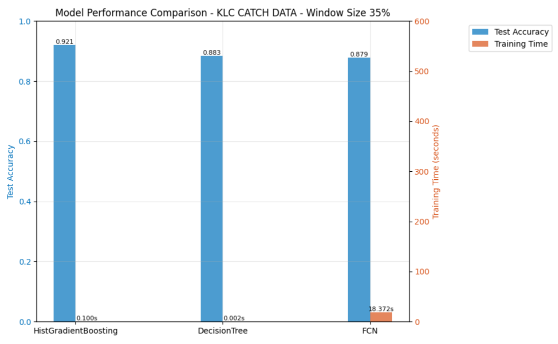
In Figure 4.2 are reported the average Test accuracies and the average Training times for the already mentioned architectures where the input were previously processed with the Catch22 algorithm.

One of the most evident effects of using Catch22 feature extraction is its impact on both accuracy and inference time. By extracting a fixed number of features for each window the inference time is kept under control and does not scale with the window length.

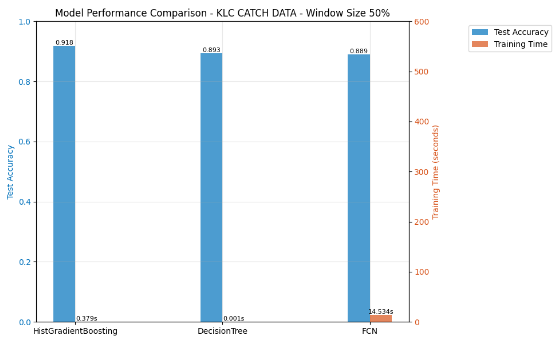
Moreover, the feature selection performed by the Catch22 algorithm proves highly beneficial, as reflected in the significant increase in the average accuracy achieved by all models. The best architecture still proves to be the HGB.



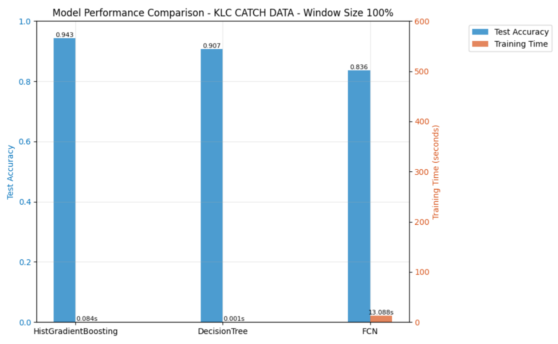
(a) Window size: 20%



(b) Window size: 35%



(c) Window size: 50%



(d) Window size: 100%

Figure 4.2: KLC performances using catch22 extraction as input

Table 4.3: Performance comparison of the models on KLC dataset using Catch22 features.

Model	Window Size	Accuracy	Training Time (s)	Prediction Time (s)
HGB	20	0.9357	0.125	3.28×10^{-5}
DT	20	0.8929	0.002	—
FCN	20	0.8886	24.402	1.43×10^{-3}
HGB	35	0.9214	0.100	3.28×10^{-5}
DT	35	0.8833	0.002	—
FCN	35	0.8786	18.372	2.30×10^{-3}
HGB	50	0.9179	0.379	9.30×10^{-5}
DT	50	0.8929	0.001	1.07×10^{-5}
FCN	50	0.8893	14.534	3.21×10^{-3}
HGB	100	0.9429	0.084	1.34×10^{-4}
DT	100	0.9071	0.001	—
FCN	100	0.8357	13.088	6.15×10^{-3}

Table 4.3 reports the overall average results obtained using catch22 features as input with the corresponding average inference times.

The use of the Catch22 features also had a significant impact on inference times (always calculated as previously). No one of the DTs' average inference times were long enough to be calculate.

The DT models stand out for their very short training times while still providing competitive accuracy, despite not being the top-performing models.

4.2 Case Study 2: Device Classification from Current Consumption

This second case study of univariate time series applications is on the "ACS-F1" dataset, made available by the AEON time series repository by Gisler et al. [16]. This dataset contains univariate time series representing the power consumption of typical appliances; the recordings are characterized by long idle periods and some high bursts of energy consumption when the appliance is active. The classes of inference correspond to 10 categories of home appliances:

1. **mobile phones** (via chargers);
2. **coffee machines**;
3. **computer stations** (including monitor);
4. **fridges and freezers**;
5. **Hi-Fi systems** (CD players);
6. **lamps** (CFL or Compact Fluorescent Lamp);
7. **laptops** (via chargers);
8. **microwave ovens**;
9. **printers**;
10. **televisions** (LCD or LED).

4.2.1 Dataset and Preprocessing

This dataset was treated similarly to the previous case as we are still dealing with univariate time series: both the windowing function (1) and the dataset preparation function (2) are the same as those shown in the Kepler Light Curves case. The parameters used here are the same; regarding the division between training and test sets, an 80-20 division was chosen in favor of the training set and the window sizes compared here are also 20%, 35%, 50% and 100% of the total size of a signal.

4.2.2 Experimental Results

Here the results of the experimental evaluation conducted on the ACS-F1 dataset are reported, as for the case before, the study investigates the impact of window size selection and feature representation on model performance, comparing tree-based methods and neural networks in terms of accuracy as well as training and inference time.

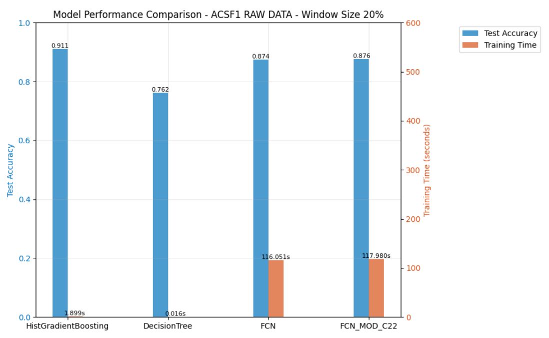
The training of the decision tree was also here performed using a grid search over the following parameters (as in the previous case study):

```
param_grid = {
    'max_depth': [3, 8, None];
    'ccp_alpha': [0.01, 0.05, 0.1].
}
```

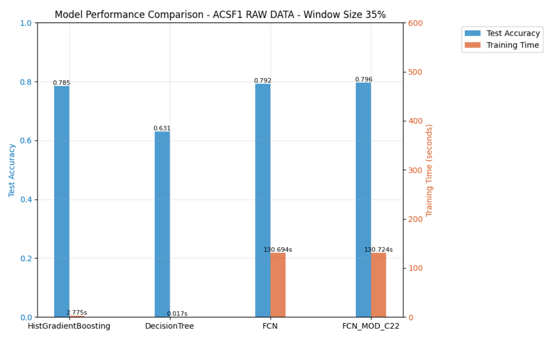
And so HGB was performed using a grid search over the following parameters (as in the previous case study):

```
param_grid = {
    'max_depth': [3, 8, None];
    'learning_rate': [0.1, 0.2, 0.3].
}
```

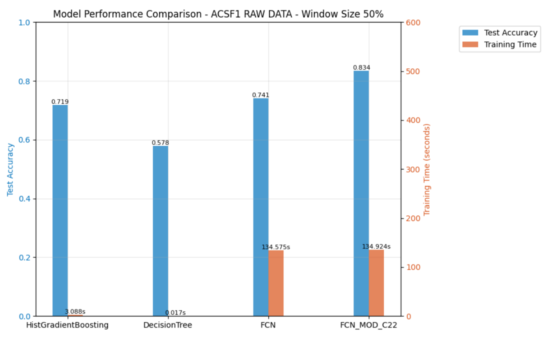
ACSF1 Dataset: Raw input Results



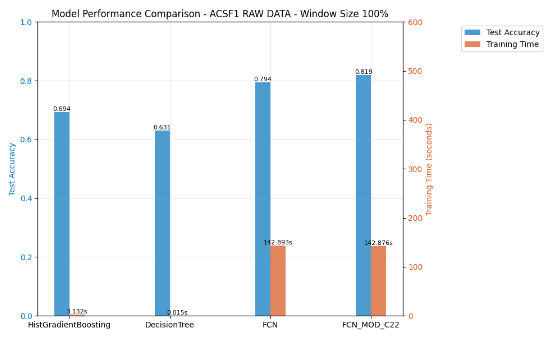
(a) Window size: 20%



(b) Window size: 35%



(c) Window size: 50%



(d) Window size: 100%

Figure 4.3: ACSF1 performances using raw time series input

Figure 4.3 reports the average test accuracies, training times, and inference times obtained on the ACSF1 dataset for the considered architectures (Decision Trees, Histogram Gradient Boosting, Fully Connected Networks, and modified FCNs) across different window sizes, using raw time series as input. As expected, the average inference times increase with the window size, since longer input sequences require more processing at prediction time.

Differently from what observed in other datasets, on ACSF1 the traditional tree-based models (DT and HGB) generally achieve lower accuracies than the neural architectures, despite being significantly faster to train, clarifying that the training times reported for the DT and HGB models correspond to the total grid search training time divided by the number of fits (45).

HGB outperforms the FCNs only in the case of the smallest window size (20%). These results suggest that tree-based models appear to achieve better performance with shorter and simpler windows. As the window size increases, all models experience a degradation in accuracy, with the worst performance observed at 100% of the time series.

Table 4.4: Performance comparison of the models on ACSF1 dataset using raw time series input.

Model	**Window Size**	**Accuracy**	**Training Time (s)**	**Prediction Time (s)**
HGB	20	0.9113	1.899	4.10×10^{-5}
DT	20	0.7625	0.016	—
FCN	20	0.8738	116.051	1.72×10^{-3}
FCN_{c22}	20	0.8762	117.980	1.72×10^{-3}
HGB	35	0.7854	2.775	6.96×10^{-5}
DT	35	0.6312	0.017	—
FCN	35	0.7917	130.694	2.83×10^{-3}
FCN_{c22}	35	0.7958	130.724	2.90×10^{-3}
HGB	50	0.7188	3.088	1.03×10^{-4}
DT	50	0.5781	0.017	—
FCN	50	0.7406	134.575	4.22×10^{-3}
FCN_{c22}	50	0.8344	134.924	4.30×10^{-3}
HGB	100	0.6937	3.132	1.85×10^{-4}
DT	100	0.6312	0.015	6.00×10^{-5}
FCN	100	0.7938	142.893	8.10×10^{-3}
FCN_{c22}	100	0.8188	142.876	8.39×10^{-3}

Table 4.4 reports the overall average results obtained using raw time series as input with the corresponding average inference times. As expected the inference time increases with the window size for all neural architectures, since longer input sequences require more computations at prediction time. Decision Trees maintain almost constant and negligible inference times across all window sizes, while HGB shows a mild but consistent increase.

ACSF1 Dataset: Catch22-Based input Results

The C22 extraction times are shown in the next table (4.5). As shown, the average feature extraction time per window increases with the window size, while the total extraction time over the entire training set decreases: this opposite trend suggests a substantial reduction in the number of training examples as the window size grows, which compensates for the higher computational cost of extracting features from longer windows.

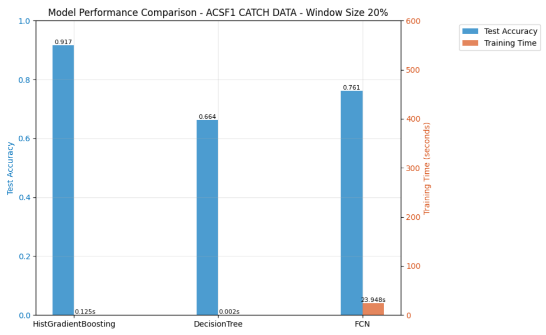
Table 4.5: Average Catch22 extraction time per sample for different window sizes on ACSF1 dataset.

Window Size (% of signal)	Tot. Ext. Time [s]	Ext. per Sample [s]
20	1.80	0.00047
35	0.32	0.00065
50	0.31	0.00092
100	0.30	0.00182

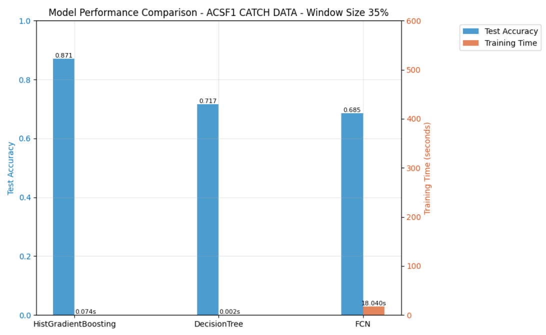
In Figure 4.4 are reported the average Test accuracies and the average Training times for the mentioned architectures where the input were previously processed with the Catch22 algorithm. For this dataset, the use of Catch22 features generally led to a decrease in classification accuracy across all tested deep architectures. A common trend can be observed for all models: the average accuracy tends to decrease as the window size increases.

This behavior is likely related to the reduction in the number of available training examples when larger windows are used. Among the evaluated models, HGB exhibits the most stable behavior, showing even better performance compared to the other architectures and maintaining an average accuracy consistently above 0.83 for 3 out 4 of the window cases, with only the Catch22 feature, while FCN_{c22} achieve the best accuracies.

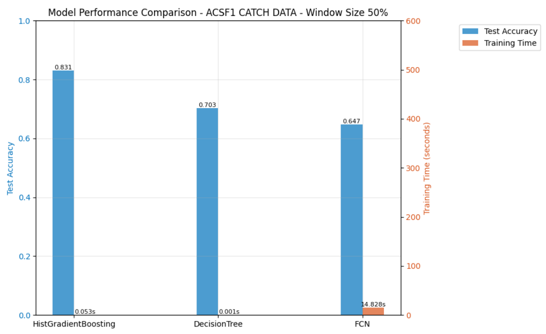
The Decision Tree displays an unusual pattern: its highest accuracy is achieved at a window size of 35%, while the lowest occurs at 20%. Afterward, its performance decreases again for both 50% and 100% window sizes. As also highlighted in the figure, the impact on training times is significant: confirming that the choice of window size strongly affects both performance and computational cost.



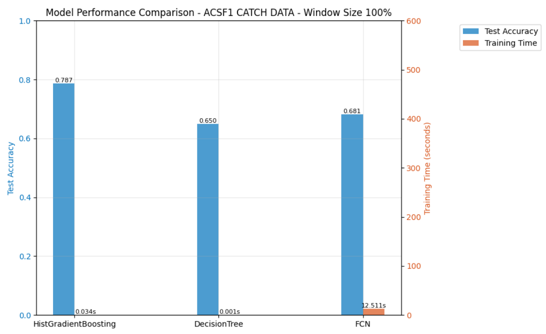
(a) Window size: 20%



(b) Window size: 35%



(c) Window size: 50%



(d) Window size: 100%

Figure 4.4: ACSF1 performances using catch22 extraction as input

Table 4.6: Performance comparison of the models on ACSF1 dataset using Catch22 features.

Model	Window Size	Accuracy	Training Time (s)	Prediction Time (s)
HGB	20	0.9175	0.125	6.94×10^{-5}
DT	20	0.6638	0.002	—
FCN	20	0.7613	23.948	1.26×10^{-3}
HGB	35	0.8708	0.074	6.31×10^{-5}
DT	35	0.7167	0.002	2.08×10^{-6}
FCN	35	0.6854	18.040	1.98×10^{-3}
HGB	50	0.8313	0.053	9.37×10^{-5}
DT	50	0.7031	0.001	3.22×10^{-6}
FCN	50	0.6469	14.828	2.92×10^{-3}
HGB	100	0.7875	0.034	2.35×10^{-4}
DT	100	0.6500	0.001	—
FCN	100	0.6812	12.511	5.34×10^{-3}

Table 4.6 reports the overall average results obtained using catch22 features as input with the corresponding average inference times. The inference times (as well as the training times) are much shorter than architectures trained on raw data, however the loss of accuracy is equally significant.

4.3 Case Study 3: Human Activity Recognition (HAR)

This case study is the only one which use a multivariate time series dataset. The MotionSense dataset has been choosed for this purpose, it has been acquired thanks to the work of Malekzadeh et al. [13], it consists of multivariate time series data acquired from accelerometer and gyroscope sensors, sampled at 20 Hz, usually embedded in smartphones; it includes measurements of attitude, gravity, user acceleration, and rotation rate collected using a mobile phone placed in the front pocket of the participants. The dataset involves 24 participants with diverse demographic characteristics, including gender, age, weight, and height where each participant performed six different activities under controlled conditions:

1. **Walking**;
2. **Jogging**;
3. **Upstairs**;
4. **Downstairs**;
5. **Sitting**;
6. **Standing**.

4.3.1 Dataset and Preprocessing

As for the other study cases the dataset has been windowed at first, but instead of the others the window sizes here are mutch shorter due to this particular case. Here, the chosen window sizes are 20, 40 and 160 time samples long since a classification task in the HAR field makes sense only if the model is able to provide predictions in a relatively short time. The selected variables include accelerometer and gyroscope signals measured along the three spatial axes (x, y, and z). The windowing process (1), and the data preparation (2), follows the same logic adopted in the previous case studies.

4.3.2 Experimental Results

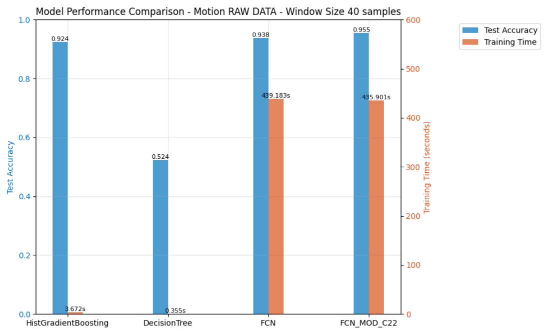
The results of the experimental evaluation conducted on the Motion Sense dataset are reported below; the study investigates the impact of window size selection and feature representation on model performance comparing tree-based methods and neural networks in terms of accuracy as well as training and inference time. The training of the decision tree was performed using a grid search over the following parameters (as in the previous cases):

```
param_grid = {
    'max_depth': [3, 8, None];
    'ccp_alpha': [0.01, 0.05, 0.1].
}
```

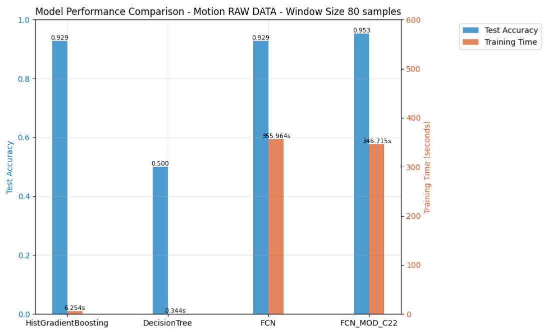
The training of the HGB was performed using a grid search over the following parameters (as in the previous cases):

```
param_grid = {
    'max_depth': [3, 8, None];
    'learning_rate': [0.1, 0.2, 0.3].
}
```

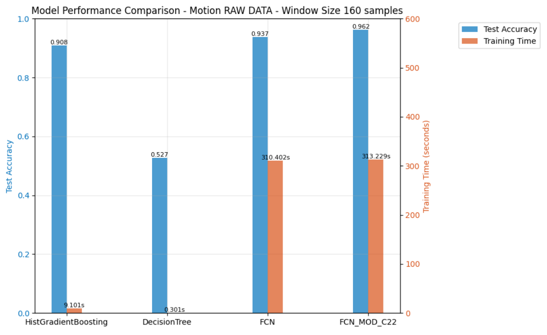
Motion Dataset: Raw input Results



(a) Window size: 40 samples (2s)



(b) Window size: 80 samples (4s)



Window size: 160 samples (8s)

Figure 4.5: Motion performances using raw time series input

Figure 4.5 reports the average test accuracies, training times, and inference times obtained on the Motion dataset using unprocessed input for the training of the architectures. The figure shows that the models remain relatively stable in terms of accuracy across the three proposed window sizes (40, 80, and 160). The FCN architectures achieve slightly higher accuracies than HGB, which remains consistently around 0.92. As observed in all experiments the training times of the FCN based models are significantly higher than those of the tree-based methods. In two out of the three window configurations (a and c), the modified FCN exhibits shorter average training times than the standard FCN despite its more complex architecture which includes the additional 22 features derived from the

Catch22 input. In terms of accuracy, the modified FCN proves to be the most accurate model for this dataset when using raw time series as input.

Table 4.7: Performance comparison of the models the Motion dataset using raw time series input.

Model	Window Size	Accuracy	Training Time (s)	Prediction Time (s)
HGB	40	0.9244	3.672	1.34×10^{-5}
DT	40	0.5243	0.355	3.53×10^{-7}
FCN	40	0.9380	439.183	1.77×10^{-4}
FCN$_{c22}$	40	0.9545	435.901	1.80×10^{-4}
HGB	80	0.9289	6.254	1.45×10^{-5}
DT	80	0.5004	0.344	6.94×10^{-7}
FCN	80	0.9286	355.964	3.15×10^{-4}
FCN$_{c22}$	80	0.9526	346.715	3.22×10^{-4}
HGB	160	0.9084	9.101	1.72×10^{-5}
DT	160	0.5272	0.301	1.01×10^{-6}
FCN	160	0.9368	310.402	5.35×10^{-4}
FCN$_{c22}$	160	0.9625	313.229	5.66×10^{-4}

Table 4.7 complements the figure by reporting the average inference times, which are not easily appreciable in the plots due to their very small values; inference time increases with the window size for all neural architectures, since longer input sequences require more computations at prediction time. In contrast, the Decision Tree maintains extremely low and nearly constant inference times across all window configurations, while HGB shows a slight but stable increase. It can also be observed that inference becomes more computationally demanding due to the multidimensional nature of the input.

Motion Dataset: Catch22-Based input Results

The C22 extraction times are shown in Table 4.8, as can be seen the extraction times follow a trend consistent with what was observed for the ACSF1 dataset. Specifically, as the window size increases the total extraction time decreases (here in a particularly pronounced manner), likely due to the strong reduction in the number of training examples. Conversely, the extraction time per window (i.e., per sample and per input) increases significantly, which in this case is largely attributable to the multidimensional nature of the signal in the Motion dataset (3 dimensions).

Table 4.8: Average Catch22 extraction time per sample for different window sizes on Motion dataset.

Window Size (n of samples)	**Tot. Ext. Time [s]**	**Ext. per Sample [s]**
40	134.41	0.00380
80	76.49	0.00433
160	45.80	0.00518

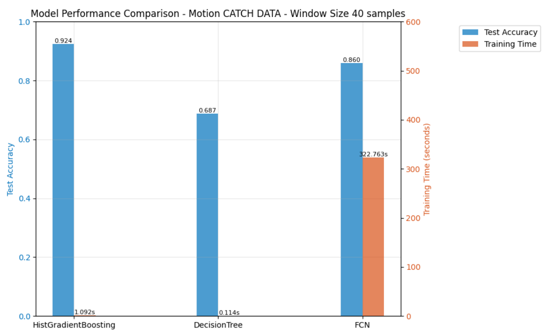
Figure 4.6 reports the average test accuracies, training times, and inference times obtained on the Motiondataset for the considered architectures (Decision Trees, Histogram Gradient Boosting, Fully Connected Networks, and modified FCNs) across different window sizes, using extracted Catch22 features as input.

The figure shows the performance trends of the models on the Motion dataset when using Catch22 synthetic features as input, compared to the raw-input setting, both HGB and DT achieve higher accuracies, while all architectures exhibit a clear reduction in training time, particularly the FCN-based models. In contrast, the FCNs, which performed better than tree-based models when using raw inputs, show a relative decrease in performance in this feature-based setting. In general, models trained on Catch22 features tend to improve as the extraction window increases,

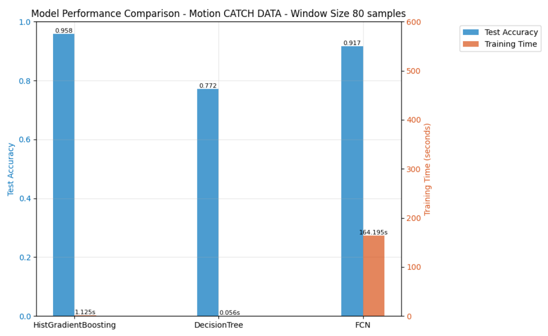
Table 4.9: Performance comparison of the models on Motion dataset using Catch22 features.

Model	**Window Size**	**Accuracy**	**Training Time (s)**	**Prediction Time (s)**
HGB	40	0.9242	1.092	9.70×10^{-6}
DT	40	0.6875	0.114	—
FCN	40	0.8595	322.763	1.41×10^{-4}
HGB	80	0.9582	1.125	1.04×10^{-5}
DT	80	0.7719	0.056	2.53×10^{-7}
FCN	80	0.9170	164.195	1.97×10^{-4}
HGB	160	0.9625	0.740	1.01×10^{-5}
DT	160	0.8265	0.029	2.83×10^{-7}
FCN	160	0.9104	90.872	3.25×10^{-4}

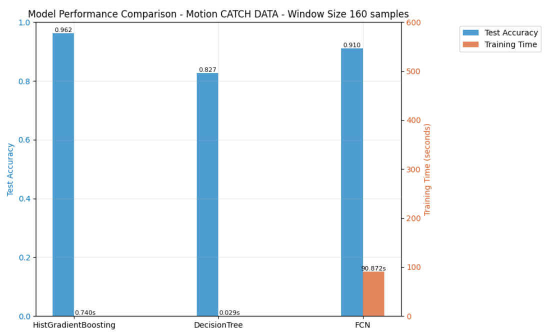
In Table 4.9 are reported the average inference times, which are not clearly visible in the corresponding figure due to their extremely small values. As shown, inference time increases with the window size for all models, reflecting the higher computational cost associated with processing longer temporal contexts. This trend is particularly evident for the FCN-based models.



(a) Window size: 40 samples (2s)



(b) Window size: 80 samples (4s)



Window size: 160 samples (8s)

Figure 4.6: Motion performances using catch22 extraction as input

In contrast, the tree-based models (DT and HGB) maintain very low inference times across all window sizes, with only a slight increase as the window length grows. Overall, these results confirm that window size has a direct impact on prediction latency and highlight the efficiency of tree-based methods at inference compared to neural architectures.

4.4 Summary of Results

From a computational perspective tree-based models and ensemble models such as the Decision Tree (DT) and Histogram-based Gradient Boosting (HGB) consistently proved to be significantly more efficient than neural networks. This superiority is evident in both training and inference phases where ensemble models maintained a drastic advantage in speed, often operating at scales that highlight the software overhead of more complex deep learning architectures.

In the KLC dataset HGB outperformed every other model, in particular the FCN by a significant margin in accuracy while completing training in a few seconds, compared to the nearly 500 seconds required by the neural network models. Regarding prediction times the disparity is even more pronounced: tree-based models exhibited near instantaneous inference, while HGB maintaining times in the order of 10^{-5} seconds, whereas FCN architectures required significantly more time, typically ranging between 10^{-4} and 10^{-2} seconds.

The experimental data reveals that these "shallow" models are not only computationally superior but also highly competitive in terms of accuracy. Interestingly, the adoption of the Catch22 algorithm appears to provide significantly greater benefits to shallow models than to standard neural networks. In both the Motion and ACSF1 datasets, the use of Catch22 features led to a substantial improvement in accuracy for the tree-based models across every temporal window tested. Conversely, the standard FCN model often experienced a performance degradation when trained on these synthetic features instead of raw data, an effect particularly pronounced in the ACSF1 case.

The experimental results suggest that while tree-based classifiers effectively leverage statistical descriptors to build robust decision boundaries, standard convolutional architectures are optimized for raw signal patterns and lose their primary advantage when the input is pre processed into a fixed set of features. The KLC dataset provides a unique case study for this transformative power of feature engineering: all models initially struggled with extremely low accuracy using raw inputs but the introduction of Catch22 features led to a dramatic improvement, more than doubling the performance across the board, with the most significant gains recorded for the tree-based models and ensemble ones.

Despite the strength of shallow models in these contexts some remarkable results were produced by the FCN$_{c22}$ architecture: by utilizing an unconventional dual input approach that combines both raw and synthetic data, specifically, the FCN$_{c22}$ achieved the highest accuracy across all window sizes when using raw input data in two of the three case studies: in the Motion and ACSF1 datasets. However, HGB model remains extremely competitive, its accuracy was slightly lower, especially when operating on Catch22 features.

Despite its increased structural complexity the FCN$c22$ demonstrated a training efficiency comparable to the standard FCN, suggesting that the additional synthetic features may facilitate a faster convergence of the loss function and may be an highly effective strategy for mastering complex time series classification tasks. In conclusion while the HGB model represents an exceptional trade off for applications requiring real time inference and low resource training, the FCNc22 stands as an interesting and robust solution.

Chapter 5

Conclusions

This thesis provides a comprehensive comparative analysis of different computational architectures for time series classification. The work began with a literature review and research phase: focusing on state of the art publications regarding Decision Trees (DT), Histogram-based Gradient Boosting (HGB), the Catch22 feature extraction pipeline and Fully Convolutional Networks (FCN). This theoretical foundation allowed for a deep study of the inherent trade offs between "shallow" tree-based models and deep learning architectures. A primary objective was to investigate the synergy between these paradigms: leading to the development of an augmented FCN architecture designed to process dual inputs.

In this modified model the traditional raw time series input is supplemented with specialized features extracted via Catch22, which are then concatenated directly to the output of the Global Average Pooling (GAP) layer. The logical thread of the thesis is built upon two main hypotheses: first to determine whether neural networks traditionally optimized for raw data can achieve superior performance when enriched with explicit synthetic features; second, to evaluate the effectiveness of tree-based models, which have significantly lower computational weight compared to neural networks.

The tree-based models HGB and DT were chosen as comparison models because, unlike neural networks, which are designed for raw data, 'shallow' tree-based models are highly effective when applied to synthetic features. The results confirm this; while the Decision Tree provided the fastest execution, the HGB model offered the best trade off between speed and precision. Notably, the HGB ensemble sometimes even has outperformed neural networks when trained on Catch22 features, proving to be a lightweight yet superior alternative in this context.

This methodological approach allowed for a comparison across distinct experimental setups, assessing how different input representations raw windows, Catch22 features, or a hybrid combination impact both classification accuracy and computational efficiency.

A critical aspect of the experimental design was the implementation of tailored windowing strategies to address the heterogeneous nature of the case studies. For the Human Activity Recognition (HAR) task fixed windows of 2, 4, and 8 seconds (40, 80 and 160 samples per window) were selected based on the temporal requirements for identifying human motion. In contrast, for the KLC and ACSF1 datasets, the windowing logic was adapted to the total length of the signals. In the KLC astronomical scenario, characterized by three month recordings with a 30 minute sampling interval, windows representing weeks of data were chosen to reflect realistic observation periods for celestial classification.

Regarding data pre processing, class imbalance was addressed through a strategic Random Under-sampling approach while operations such as data splitting and normalization were conducted using standard library functions, specifically StandardScaler: to ensure statistical consistency and experimental reproducibility across all tested architectures.

The integration of Catch22 features into the Deep Learning pipeline underwent several design iterations. In this finalized architecture the feature extraction is conducted offline, relative to the main classification process, significantly streamlining the training phase. The second input vector of 22 deterministic features is concatenated directly to the output of the Global Average Pooling (GAP) layer.

This design choice is technically motivated by the nature of the GAP layer itself; unlike convolutional layers, the pooling layer contains no trainable parameters and operates as a deterministic function [7]. By appending the Catch22 vector at this specific stage, the model effectively treats these 22 external features as if they were high level representations extracted by the network's own previous layers.

The motivation for comparing established architectures with the proposed dual input FCN lies in the exploration of a hybrid paradigm that merges deep learning capabilities with traditional feature engineering. Usually a fundamental trade off exists between classical machine learning models and neural networks. While traditional models rely heavily on complex feature extraction and selection techniques, neural networks are designed to operate directly on raw signal data. Although neural networks are more computationally expensive to train, they offer significantly greater generalization power, an advantage given that available datasets are often smaller than what is ideally required for complex tasks.

One of the target of this work was to investigate whether providing a neural network with both raw input and synthetic features could bridge the gap between these two approaches. While synthetic features, such as those provided by AEON, are conventionally used with simpler machine learning algorithms, their integration into an FCN allows the model to leverage precomputed statistical descriptors alongside its own internal feature learning process. By bypassing the need for a separate, heavy feature selection phase while still benefiting from domain specific descriptors, this hybrid approach seeks to enhance performance in contexts where raw data alone might not fully capture the underlying temporal dynamics, and lead to the possibility of defining a battery of synthetic feature extraction strategies directly within the architecture. This shifts the focus from manual feature engineering to architectural design, where the selection of relevant features is inherently driven by the network's structure.

Moreover, this strategy tests the hypothesis that the superior generalization of deep learning can be further refined by the explicit injection of established time series characteristics, potentially leading to more robust classification outcomes. The proposed dual input FCN architecture demonstrated a clear advantage over the standard FCN in most of the experimental scenarios: the integration of synthetic Catch22 features alongside raw time series consistently led to improved classification performance, indicating that neural networks can be positively influenced by the explicit injection of handcrafted domain descriptors through a secondary input branch.

In two out of the three datasets (ACSF1 and Motion) the FCN_{c22} architectures achieved higher accuracies than the tree based models and showed greater resilience to the reduction in the number of training examples caused by larger window sizes, these findings could suggest that while traditional models benefit from engineered features and computational efficiency, neural architectures are better suited to capture complex temporal patterns when enriched with complementary statistical information.

Importantly, this performance improvement does not come at the cost of a substantial increase in computational overhead. Compared to the standard FCN, the modified architecture exhibits similar inference and training times with improved recognition accuracy in most of the investigated case studies.

Bibliography

[1] Valerio Freschi. Machine learning. Materiale didattico, Università degli Studi di Urbino Carlo Bo, 2023. Anno Accademico 2024/2025, Corso di Laurea in Informatica e Innovazione Digitale.

[2] Scikit-learn developers. Decision trees, accessed: 2024-12-17 . `https://scikit-learn.org/stable/modules/tree.html`, 2024.

[3] Scikit-learn developers. Ensemble trees, accessed: 2024-05-22 . `https://scikit-learn.org/stable/modules/ensemble.html`, 2024.

[4] Scikit-learn developers. Random forest, accessed: 2024-12-17 . `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`, 2024.

[5] Scikit-learn developers. Hist gradient boosting, accessed: 2024-12-17 . `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`, 2024.

[6] Liqian Zhou, Jiao Song, Zejun Li, Yingxi Hu, and Wenyan Guo. Thgb: predicting ligand-receptor interactions by combining tree boosting and histogram-based gradient boosting. *Scientific Reports*, 2024.

[7] Sara Montagna. Applicazioni dell'intelligenza artificiale - biologia e medicina. Materiale didattico, Università degli Studi di Urbino Carlo Bo, 2024. Anno Accademico 2024/2025, Corso di Laurea in Informatica e Innovazione Digitale.

[8] Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. *CoRR*, 2016.

[9] Hassan Ismail Fawaz, Germain Forestier, François Petitjean, and Frédéric Denis. Fcn by fawaz et al., accessed: 2024-12-17 . `https://github.com/hfawaz/dl-4-tsc/blob/master/classifiers/fcn.py`, 2019.

[10] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 2019.

[11] Carl H. Lubba, Sarab S. Sethi, Philip Knaute, Simon R. Schultz, Ben D. Fulcher, and Nick S. Jones. catch22: Canonical time-series characteristics. *Data Mining and Knowledge Discovery*, 2019.

[12] Aeon developers. Aeon time series dataset repository, accessed: 2024-12-17 . `https://timeseriesclassification.com`, 2023.

[13] Mohammad Malekzadeh. Motionsense dataset, accessed: 2025-12-27 . `https://www.kaggle.com/datasets/malekzadeh/motionsense-dataset`, 2019.

[14] Nicholas H Barbara, Timothy R Bedding, Ben D Fulcher, Simon J Murphy, and Timothy VanReeth. Classifying kepler light curves for 12000 a and f stars using supervised feature-based machine learning. *Monthly Notices of the Royal Astronomical Society*, 2022.

[15] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 2017.

[16] Christophe Gisler, Antonio Ridi, Damien Zufferey, Omar Abou Khaled, and Jean Hennebert. Appliance consumption signature database and recognition test protocols. In *2013 8th International Workshop on Systems, Signal Processing and their Applications (WoSSPA)*, 2013.

[17] Marília Barandas, Duarte Folgado, Letícia Fernandes, Sara Santos, Mariana Abreu, Patrícia Bota, Hui Liu, Tanja Schultz, and Hugo Gamboa. Tsfel: Time series feature extraction library. *SoftwareX*, 2020.

[18] Liqian Zhou, Jiao Song, Zejun Li, Yingxi Hu, and Wenyan Guo. Thgb: predicting ligand-receptor interactions by combining tree boosting and histogram-based gradient boosting. *Scientific Reports*, 2024.

[19] Simone Ludwig. Optimization of control parameter for filter algorithms for attitude and heading reference systems. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018.

Acknowledgments

I would like to express my gratitude to my family for their constant support.
I am also deeply thankful to my supervisor, who not only guided me throughout the writing of this thesis, but was also an outstanding professor of Machine Learning.
Finally, I would like to thank myself for my perseverance and dedication.