UNIVERSITY OF CALIFORNIA
RIVERSIDE

Accelerating Streaming Time Series Feature Extraction Using an FPGA

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Prithviraj Yuvaraj

September 2022

Thesis Committee:

Dr. Philip Brisk, Chairperson
Dr. Eamon Keogh
Dr. Daniel Wong

The Thesis of Prithviraj Yuvaraj is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Philip Brisk, without whose help, I would not have been here. I would also like to mention the assistance my peer, Amin Kalantar, provided in regards to the technical portions of this work.

To my parents and brother for all their support.

ABSTRACT OF THE THESIS

Accelerating Streaming Time Series Feature Extraction Using an FPGA

by

Prithviraj Yuvaraj

Master of Science, Graduate Program in Computer Science
University of California, Riverside, September 2022
Dr. Philip Brisk, Chairperson

With the increase in available data, specifically time series data, the importance of different data analysis techniques has increased. One technique used by many data scientists is finding characteristics within subsequences of the data set. Characteristics, or features can be quantified by a process known as feature extraction . This feature extraction step is often computationally expensive, and usually requires the availability of the entire data set. During the data collection stage, data scientists may want to see patterns or characteristics. This can be achieved with Real-Time Feature Extraction, by computing feature sets while data streams in from a source. FPGAs, or Field Programmable Gate Arrays, have access to a plethora of I/O that allows for data to be streamed directly into the computational units making for efficient,real-time feature extraction. In this paper, we provide an FPGA architecture that is able to extract features in real-time, while offering latency, and power optimizations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the wide adoption of IoT devices, the amount of data, specifically time series data, available has increased exponentially. The need for efficient analysis methods has risen, which gives way to the advent of hardware accelerating data analysis techniques. A prominent technique performed on time series data consists of extracting features to quantify characteristics within the data set. Features can then go onto be used in searches, classification, machine learning models, etc. Although features strip away many of the details within the data, data analysis techniques, such as classification, using feature-based time series representations have known to out perform other shape-based or dictionary-based methods [15].

One specific application that used feature-based analysis is NLP Time Series Search, which used natural language processing to search subsequences within a time series that match a user specified phrase [12]. The feature extraction stage of the application is computationally expensive. These feature extractors were implemented in Matlab and Python, which are not thought of as performance optimal languages. Many feature extraction algorithms require the entirety of the data set be known, and data analysis techniques occur well after the data collection phase. A system where data points are analyzed during the collection stage would allow for scientists to learn the characteristics of the data in real-time. Feature extraction algorithms should provide performance optimal, real-time computation which can be achieved by using an FPGA (Field Programmable Gate Arrays).

Traditional CPU systems have to pass data through a memory hierarchy before computation begins. An FPGA is able to directly stream data points onto the compute architecture, as well as offers a plethora of I/O. The combination of these two traits allow FPGAs to be a perfect candidate to perform real-time feature extraction. FPGAs can also leverage parallelism to reduce latency of computation, while in general running at lower clock frequencies resulting in lower power consumption. High-Level Synthesis (HLS) tools make the process of creating FPGA architecture simpler by converting software code into register-transfer language (RTL), and allow for design to be tested in a timely fashion.

In this paper, we provide an a streaming feature extraction accelerated using an FPGA. The design is able to extraction 25 features from data that is streamed directly into the hardware. The features were selected from NLP Time Series Search [12]. The feature extraction algorithms were optimized for streaming input and converted to RTL using Xilinx's Vitis HLS 2020.2. The design was then tested and performance analysis was conducted trying to improve bottlenecks. The rest of the paper is organized as follows.

We initially offer some background on feature extraction and time series analysis in general, as well as high-level synthesis and hardware acceleration using an FPGA. Then definitions are given to explain the algorithms thoroughly. After the feature extraction algorithms are defined, the hardware implementation is shown. We then provide the results of our prototype kernel. Finally, conclude with some areas the research can be continued.

Chapter 2

Related Works

2.1 Feature Extraction in Time Series Analysis

Due to the increased adoption of the Internet of Things, a large amount of time series data has emerged. Time series data is a measurement, observation, or estimate ordered in time. There are multivariate time series which add multiple measurements of data over time, and spatial-temporal time series that take into account the spatial domain. The most common time series is an univariate time series that has one value for each time step. Multivariate time series can be seen in Principal Component Analysis and Hidden Markov Models [20], and uni-variate feature-based time series analysis is used for subsequence searches [12].

Analysis can be performed on the data sets that allow for trends, motifs, or features to be extracted to characterize the time series data. Many methods of analysis have been developed for both univariate [8] and multivariate [20] time series data.Characterization of time series has become an important analysis technique as they are crucial components of applications from a wide field of study. Feature-based time series analysis is argued to outperform shape-based and dictionary-based representations in classifications taskts [15]. Different methods to extract features from the time series data have been explored to reduce the expense of the analysis such as finding the most useful, influential features to reduce the size of the feature set. A few standard feature sets [15] have emerged that have been thoroughly evaluated in comparative studies [8].

Another method used in time series classification is Dynamic Time Warping (DTW) which finds similarities in temporal data sequences. DTW is an extensively explored area and many optimizations have been proposed [23]. These optimizations usually fall into two categories, algorithm modification or hardware acceleration. The DTW algorithm has been improved by adding weights to make a Weighted DTW (WDTW) which can be used to penalize outliers improving performance on feature detection. The DTW algorithm has been accelerated with different hardware such as GPUs, and FPGAs [19] [21].

2.2 Hardware Accelerated Software Application

Hardware can be used to improve the performance of different workloads by exploiting parallelism within the software applications. Different hardware such as

FPGAs and GPUs are capable and proficient at improving performance by acting on the inherit parallelism. FPGAs achieve parallelism by using multiple processing engines (PEs), a portion of the design that does the computation, to perform more of the computation simultaneously. Timeseries analysis techniques have been ported over to FPGA such as DTW, Matrix Profile, and other simpler tecniques like outlier detection [16].

The prevalence of time series data sets and subsequence searches have given rise to different techniques to speed up performance. One of these techniques is using hardware accelerators to speed up computation times such as FPGAs. A popular time series analysis method is Dynamic Time Warping (DTW) which has been extensively explored and ported to FPGA architecture. DTW is a method of time series analysis used to measure similarities between sequences of data [23]. As a very popular form of time series analysis many different optimizations and implementations were explored using FPGA architecture [19, 21, 14].

Another time series analysis technique is using a Matrix Profile. The Matrix Profile is computed to discover anomalies and trends with in the time series data, many different methods have been created and compared that efficiently solve for MP such as SCAMP, STAMP, SCRIMP, etc [24]. Python is often used in these implementation, which might not be the most efficient and optimal design choice. Efforts have been made to port SCAMP to a heterogeneous CPU + FPGA [18], and alternative method of computing the Matrix Profile using an FPGA was created using HLS tools has also been explored [13].

2.3 HLS in FPGA Development

A majority of FPGA development is done with Hardware Design Languages (HDLs) such as VHDL or Verilog. These languages are often tricky to properly develop with leading to long development cycles. High-Level Synthesis is a method that allows for quick FPGA implementations to be created by using high-level programming languages to develop usable RTL. This FPGA architecture was created using Xilinx's Vivado/Vitis HLS which is C/C++ based. Research has been done to explore different aspects of HLS tools from improvements in synthesis of HLS code, optimizations in the generated RTL, and even expand the type of programming languages that have HLS support.

HLS synthesis, placing, and routing routines often take a large portion of time, that causes development of FPGA architectures to slow down. Efforts have been made to improve performance of HLS compilation by exploiting parallelism within the compilation workload [10]. Multi-core CPUs or GPUs running newly created parallel routing, and placement algorithms were used to reach fast compilation time.

Another avenue of research is the quality of the HLS synthesized hardware. Often experienced FPGA developers can create more efficient designs than the HLS counterparts. Research has been explored on improving the performance of the output architecture created by HLS designs. A study was conducted about improving the data placement to reduce the latency of data related operations, that can occur when kernels try to use the wrong type of memory structures [22]. Design improvements can also be made with regard to the scheduling on instructions which can improve timing regarding cycles in HLS code [4, 3]. Efforts have also been made to add different high-level languages that have HLS FPGA development support such as PyLog [11]. Instead of worrying about the architectural specifics developers can focus on the algorithmic details and performance.

Chapter 3

Background

3.1 NLP Time Series Search

NLP Time Series Search is a method to use natural language to search time series sequences within a data set. The search is performed by computing features scores, or values that describe a characteristic that is present within the sequence of values, that are then matched with words [12]. The feature scores are computed using feature extraction algorithms that analyze a subsequence of the time series data and output a single scalar value that acts as a rating of the characteristic within that subsequence. There are 27 characteristics, or from henceforth called features, that are originally computed in NLP Time Series Search. Before feature extraction, a time series data set must be known in its entirety and then scores are computed for each feature using subsequences of a predetermined length. Once the feature scores for the entire time series data are computed, a feature specific normalization occurs across each feature. The newly normalized features are used in the time series search.

3.2 Definitions

We begin with some preliminary definitions.

Time Series: A time series T of length n is a sequence of scalar numbers t_i : $T = \langle t_1, t_2, \ldots t_n \rangle$. For the purpose of expository discussion, we assume that all time series datapoints are real numbers ($\mathbb{R}$).

Subsequence: A subsequence is a contiguous subset of the datapoints in T. typically specified by starting index i and length m: $T_{i,m} = \langle t_i, t_{i+1}, \ldots, t_{i+m-1} \rangle$.

When appropriate, we will denote a time series as a real-valued vector of length n ($\mathbb{R}^n$). While a time series has a specific connotation (i.e., datapoints taken in order, often with a known sampling interval), a time series can be treated as a vector when appropriate. All operations that can be applied to vectors can just as easily be applied to time series. We denote the **origin** (a vector of all zeroes) as $\mathbf{0}$; if the dimension is needed, we denote it as a subscript, i.e., $\mathbf{0}_n$; likewise, we denote a vector of all ones as $\mathbf{1}$ or $\mathbf{1}_n$.

3.3 Norms

A **norm** is a function $\mathcal{M} : \mathbb{R}^n \to \mathbb{R}^{\geq 0}$ that maps a vector to a non-negative valued real-valued number, and satisfies the following properties:

1. Triangle Inequality: $\mathcal{M}(X + Y) \leq \mathcal{M}(X) + \mathcal{M}(Y) \ \forall \ X, Y \in \mathbb{R}^n$.

2. Absolute Homogeneity: $\mathcal{M}(sX) = s\mathcal{M}(X) \ \forall \ s \in \mathbb{R}, X \in \mathbb{R}^n$.

3. Positive Definiteness: $\mathcal{M}(X) = 0 \iff X = \mathbf{0} \ \forall \ X \in \mathbb{R}^n$.

In this thesis, we are primarily concerned with two specific norms, defined as follows:

***L^1* Norm:** The L^1 norm of a vector $X \in \mathbb{R}^n$, denoted $\|X\|_1$, is the sum of the absolute values of the scalar datapoints in X:

$$\|X\|_1 = \sum_{i=1}^{n} |x_i| \tag{3.1}$$

***L^2* Norm:** The L^2 norm of a vector $X \in \mathbb{R}^n$, denoted $\|X\|_2$, is the magnitude of X, i.e., the distance of the point $\langle x_1, x_2, \ldots, x_n \rangle \in \mathbb{R}^n$ from the origin in an n-dimensional Euclidean space:

$$\|X\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{3.2}$$

3.4 Normalization

Normalization is the application of a function $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^n$, which adjusts the value of each scalar within a vector to a common scale, typically $[0, 1]$ or $[-1, 1]$, in a manner that preserves key qualitative and/or quantitative properties. In the context of a time series, normalization may preserve the points which are local and global minima, subsequence shapes, etc.; different normalization methods may be applied as pre- and/or post-processing steps when performing computations on time series.

In the following, let μ and σ denote the arithmetic mean and standard deviation of a vector $X \in \mathbb{R}^n$ and let x_{min} and x_{max} denote the maximum scalar values in X.

z-score Normalization ($\mathcal{N}_z$) computes a vector $Z = \mathcal{N}_z(X) \in \mathbb{R}^n$ with entries

$$z_i = \frac{x_i - \mu}{\sigma}. \tag{3.3}$$

Min-Max Normalization computes a vector $Y = \mathcal{N}_{min-max}(X) \in \mathbb{R}^n$, with entries:

$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}. \tag{3.4}$$

Min-Max Normalization with Minimum Transform computes a vector $Y = \mathcal{N}_{min-max}^{min-trans}(X) \in \mathbb{R}^n$, with entries:

$$y_i = 1 - \frac{x_i - 2x_{min}}{x_{max} - x_{min}}. \tag{3.5}$$

Min-Max Normalization Scaled by the Maximum computes a vector $Y = \mathcal{N}_{min-max}^{max-scaled}(X)$, with entries:

$$y_i = \frac{x_i - x_{min}}{x_{max}^2 - x_{min}x_{max}}, \tag{3.6}$$

or, equivalently: $\mathcal{N}_{min-max}^{max-scaled}(X) = \frac{1}{x_{max}} \cdot \mathcal{N}_{min-max}(X)$.

Min-Max Normalization with Positive Guarantee computes a vector $Y = \mathcal{N}_{min-max}^{+}(X)$, with entries:

$$y_i = \begin{cases} \frac{x_i - x_{min}}{x_{max} - x_{min}} & x_i \geq 0 \\ \\ \frac{x_i + x_{max} - 2x_{min}}{x_{max} - x_{min}} & x_i < 0 \end{cases} \tag{3.7}$$

Both cases can be rewritten in terms of Min-Max Normalization:

- Case $x_i \geq 0$ returns the i^{th} scalar obtained from $\mathcal{N}_{min-max}(X)$
- Case $x_i < 0$ returns the i^{th} scalar obtained from $\mathcal{N}_{min-max}(X + (x_{max} - x_{min}) \cdot \mathbf{1}_n)$.

3.5 Features and Feature Extraction

A **feature** is defined as a computable property of a time series. We compute features for every length-m subsequence of a time series, which we collect into a “meta-time series” called a **feature vector**. **Feature extraction** refers to the process of computing a feature vector by an algorithm called a **feature extractor**. By convention, we require all feature vectors to be normalized as a post-processing step. Many different features can be defined and computed for a time series concurrently, and it is common to define specific features in terms of other features. Our implementation of NLP Time Series search computes 25 features. Feature extraction is a two step process, consisting of scoring followed by normalization.

Scoring: A **scoring function** $f : \mathbb{R}^m \rightarrow \mathbb{R}$ maps a subsequence to a scalar value s called a **(feature) score**. Computing scores over all sequences of a time series yields a meta-time series called a **score vector**: $S = \langle s_1, s_2, \ldots, s_{n-m+1} \rangle$, where $s_i = f(T_{i,m})$. The score vector F is shorter than the original time series T because subsequences starting at indices higher than $n - m + 1$ have fewer than m datapoints, and their features are not computed.

Normalization: A feature vector is a normalized score vector $F = \langle r_1, r_2, \ldots, r_{n-m+1} \rangle$, which is computed as follows:

$$F = \mathcal{N}(S) = \mathcal{N}(\langle f(T_{1,m}), f(T_{2,m}), \ldots, f(T_{n-m+1,m}) \rangle). \tag{3.8}$$

Each normalized datapoint r_i in the feature vector F corresponds to unnormalized score s_i in the score vector S.

Feature Pairs: Let $F = \langle 1-r_1, 1-r_2, \ldots, 1-r_{n-m+1} \rangle$ and $F' = \langle r'_1, r'_2, \ldots, r'_{n-m+1} \rangle$ be feature vectors. F and F' **paired** if $F' = \mathbf{1}_{n-m+1} - F$, i.e., if $r'_i = 1 - r_i$,

$1 \leq i \leq n - m + 1$. Rather than computing both F and F' from first principles, it is often more efficient to compute F directly and then derive F' from F (or vice-versa).

When appropriate, we refer to the scoring function and feature vectors by the name the feature, e.g., $f_{Nonlinearity}$ and $F_{Nonlinearity}$ for a feature named "Nonlinearity." This nomenclature omits the name of the normalization method chosen for the "Nonlinearity" feature, but simplifies the discussion significantly.

3.6 Streaming NLP Time Series Search

While offline analyses typically assume that the entire time series is stored and available for processing, our work considers time series in the streaming context, where only the most recent window of data points is available. The exact length of the time series (n) being streamed is not known in the general case, and the time series is implicity treated as having infinite length.

Streaming Time Series: For streaming, we redefine the time series T as a window of length $w << n$, i.e., $T = \langle t_1, t_2, \ldots, t_w \rangle$; t_1 is the oldest datapoint in the window, while t_w is the most recently sampled datapoint. When a new datapoint is sampled during streaming, t_1 is discarded from the window. Certain computations may involve both t_1 and the newly sampled datapoint, which we denote as t_{w+1}, and t_1 can only be discarded *after* performing those computations. After discarding t_1, we adjust the indices of the datapoints in the window so that $t_i \rightarrow t_{i-1}, 2 \leq i \leq w+1$, so that the newly sampled datapoint becomes t_w at index w. A typical implementation in either hardware or software is to use a circular buffer.

Streaming Feature Extraction: A streaming feature extractor generates one scalar score value $s = f(T)$ from the current window and transmits this value to the streaming normalizer, defined below.

Streaming Normalizer: The streaming normalizer buffers incoming score values in batches of length w, normalizes each batch, and then outputs the normalized batch in a burst of size w. To simplify notation, we denote the **score buffer** as $S = \langle s_1, s_2, \ldots, s_w \rangle$ and the (normalized) **streaming feature vector** as $F = \mathcal{N}(S) = \langle r_1, r_2, \ldots, r_w \rangle$.

3.7 Streaming vs. Offline NLP Time Series Search

Suppose that we stream a time series T of length n through the streaming NLP Time Series Search algorithm. Without loss of generality, assume that $n = kw$ for positive integer k, which ensures that the length of T is a multiple of the window size. The score buffer computed when processing the j^{th} window of datapoints during streaming is the subsequence $S_{(j-1)w+1,w}$ of the score vector S that would be computed offline. Streaming normalization then produces a streaming feature vector $\mathcal{N}(S_{(j-1)w+1,w})$.

By normalizing on the granularity of window sizes, the **streaming normalization function**, $\mathcal{N}_{Stream}$, produces the following normalized word feature vector, in which subsequences of length w have been independently normalized.

$$\mathcal{N}_{Stream}(S) = \langle \mathcal{N}(S_{1,w}), \mathcal{N}(S_{w+1,2w}), \ldots, \mathcal{N}(S_{(k-1)w+1,kw}) \rangle. \tag{3.9}$$

In contrast, the offline NLP Time Series Search normalizes all data points in the word feature vector F. Therefore, in the general case, one cannot assume that $\mathcal{N}(S)$ and $\mathcal{N}_{Stream}(S)$ will produce the same output.

Chapter 4

Feature Extraction Algorithms

Our implementation of NLP Time Series Search extracts 25 features, 22 of which are paired. For each feature, the input is a time series window $T = \langle t_1, t_2, \ldots, t_w \rangle$. The feature extractor first computes a scoring function which produces a scalar score value s, which is stored in a score buffer S of length w. When S is full, the streaming normalizer outputs a streaming feature vector $F = \mathcal{N}(S)$. Table 4.1 lists all of the features that are computed, including pairing information, and specifies the normalization method that is applied to the score buffer computed for each feature.

This chapter specifies the scoring function and normalization method used to compute each feature; illustrative examples are shown for each feature using time series that are publicly available in the UCR Time Series Archive [7].

Table 4.1: Feature Pairs

Feature Pairs		Normalization Method
Low	-	$\mathcal{N}_{min-max}^{min-trans}$
High	-	$\mathcal{N}_{min-max}$
High Amplitude	Low Amplitude	$\mathcal{N}_{min-max}$
Spike	Dropout	$\mathcal{N}_{min-max}$
Rising	Falling	$\mathcal{N}_{min-max}^{max-scaled}$
Complex Normalize	Simple Normalize	$\mathcal{N}_{min-max}$
Complex Unnormalize	Simple Unnormalize	$\mathcal{N}_{min-max}$
Symmetric	Asymmetric	$\mathcal{N}_{min-max}^{+}$
Linearity	Nonlinearity	$\mathcal{N}_{min-max}$
Convex	Concave	$\mathcal{N}_{min-max}$
Noise	Smooth	$\mathcal{N}_{min-max}$
Step	No Step	$\mathcal{N}_{min-max}^{+}$
Periodic	Aperiodic	$\mathcal{N}_{min-max}$
Unique Patterns	-	$\mathcal{N}_{min-max}$

4.1 Low

The **Low** feature identifies subsequences in a time series having a small sum of values relative to the time series as a whole; Figure 4.1 shows an example. Low maintains a running sum s of all of the datapoints in the current window T. Each time a new datapoint is sampled, the sum is updated:

$$s = s + t_{w+1} - t_1 \tag{4.1}$$

The feature score $f_{Low}(T) = s$ is then transferred to the normalizer. Low employs a unique normalization method (not used by any other feature extractor) that implements much of its functionality. When a complete score buffer S_{Low} becomes available, it is normalized: $F_{Low} = \mathcal{N}_{min-max}^{min-trans}(S_{Low})$ and then output in a burst of size w.

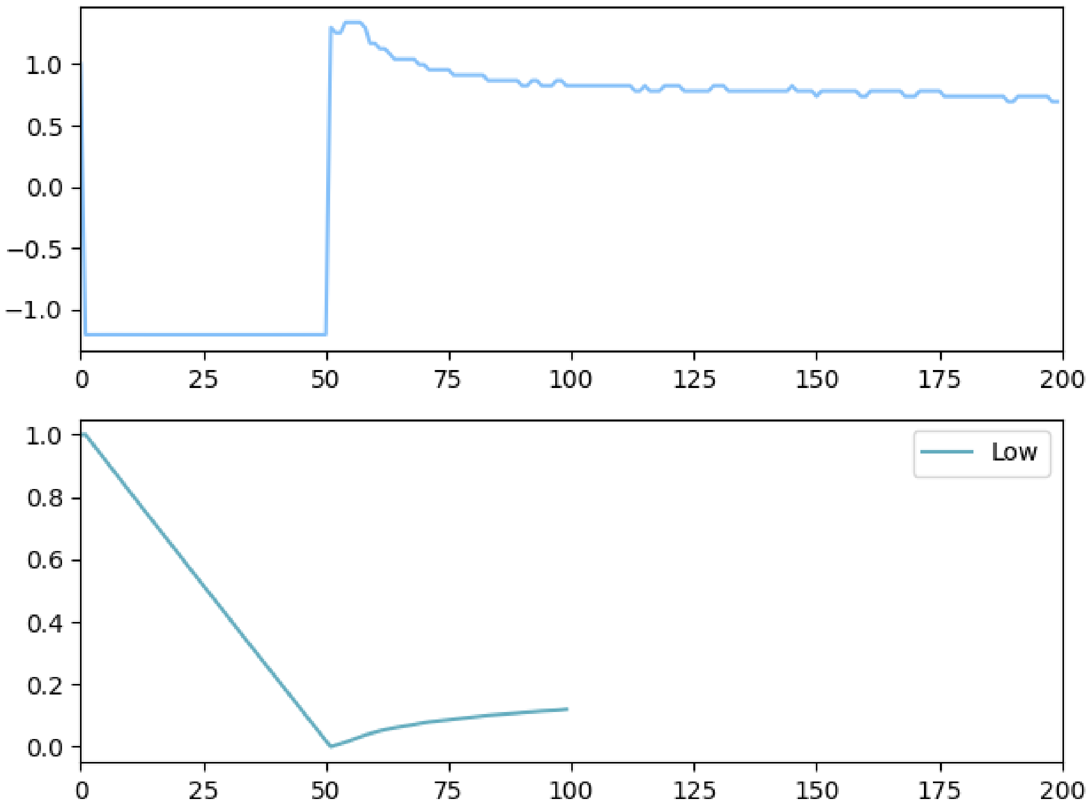


Figure 4.1: **Low:** An example of the Low feature computed for the Freezer dataset using a window of size $w = 100$; the Low feature is not computed for the last 100 datapoints.

4.2 High

The **High** feature classifies sub-sequences of a longer time series, that, on average have larger values than others. Figure 4.2 shows an example. Similarly to Low, a running sum is used to compute the arithmetic mean μ of the datapoints in the current window T.

$$\mu = \mu + \frac{t_{w+1}}{w} - \frac{t_1}{w}. \tag{4.2}$$

The feature score $f_{High}(T) = \mu$ is then transferred to the normalizer. When a complete score buffer S_{High} becomes available, it is normalized: $F_{High} = \mathcal{N}_{min-max}(S_{High})$ and output in a burst of size w.

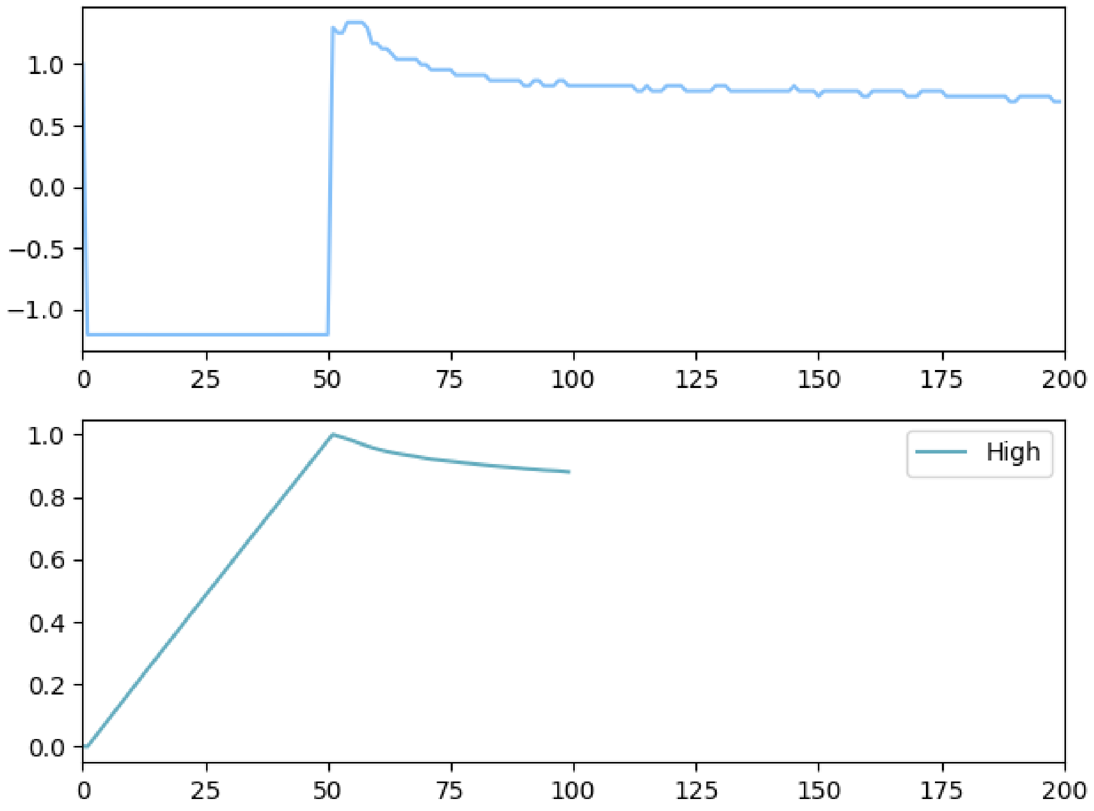


Figure 4.2: **High:** An example of the High feature computed for the Freezer dataset using a window of size $w = 100$; the High feature is not computed for the last 100 datapoints.

4.3 High and Low Amplitude

The **High** and **Low Amplitude** features compute the standard deviation of the current window T; Figure 4.3 shows an example.

The arithmetic mean μ can be updated for each subsequence using Eq. 4.2, and the standard deviation σ is then computed as:

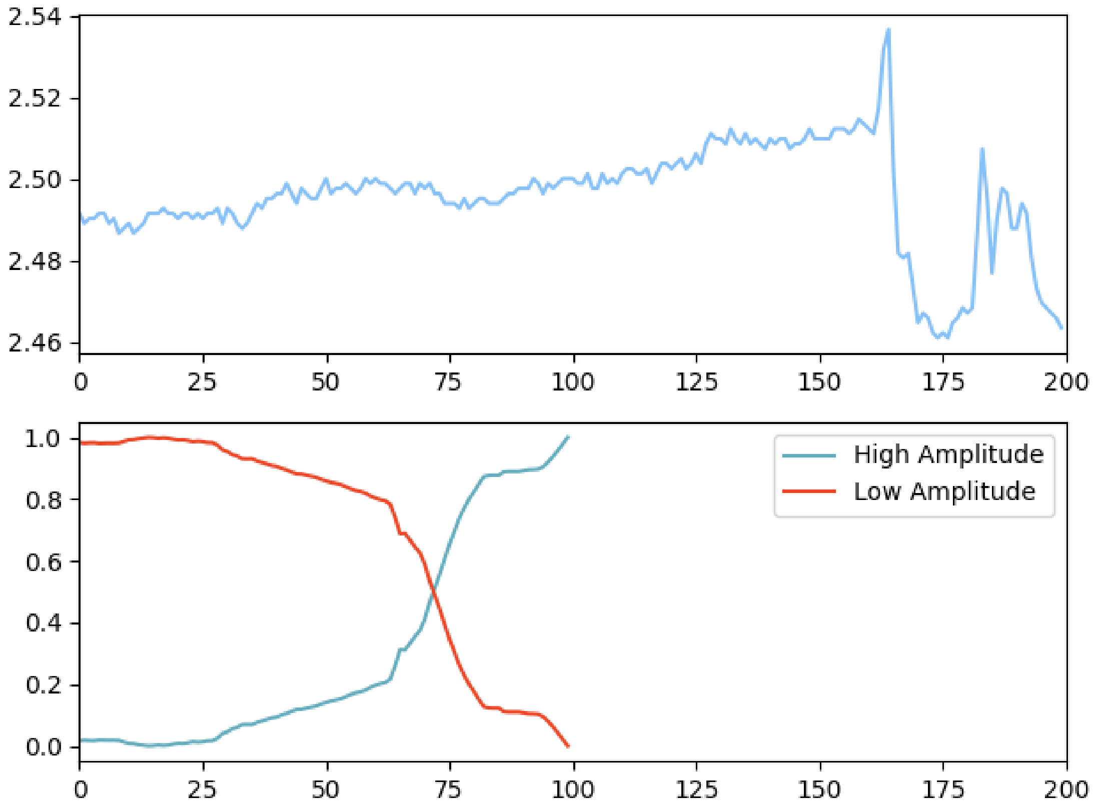


Figure 4.3: **High and Low Amplitude:** An example of the High and Low Amplitude features computed for the Insect EPG dataset using a window of size $w = 100$; the High and Low Amplitude features are not computed for the last 100 datapoints.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{w}(t_i - \mu)^2}{w}} \tag{4.3}$$

The feature score $f_{High-Amplitude}(T) = \sigma$ is then transmitted to the normalizer. When a complete score buffer $S_{High-Amplitude}$ becomes available, it is normalized: $F_{High-Amplitude} = \mathcal{N}_{min-max}(S_{High-Amplitude})$ and output in a burst of size w. The normalized Low Amplitude feature vector can be derived from the normalized High Amplitude feature vector: $F_{Low-Amplitude} = \mathbf{1}_w - F_{Low-Amplitude}$.

4.4 Spike and Dropout

The **Spike** and **Dropout** features quantify the maximum difference between the median and any other datapoints within the current window T; Figure 4.4 shows an example. A function $Median : \mathbb{R}^w \to \mathbb{R}$ computes the median of T; a function $MedianFilter : (\mathbb{R}^w, \mathbb{N}) \to \mathbb{R}^w$ applies a median filter to T, yielding a vector $D^{(k)} = \langle d_1^k, d_2^{(k)}, \dots d_w^{(k)} \rangle = MedianFilter(T, k)$, where $d_i^{(k)} = Median(t_{i-k}, t_{i-k+1}, \dots t_{i+k})$; when applying the median filter, all entries in T having indices less than 1 or greater than w are set to zero. The Spike feature score is then computed as the maximal difference among the datapoints of T and their median filtered values:

$$f_{Spike}(T) = \max_{1 \le i \le w} |t_i - d_i^{(1)}| \tag{4.4}$$

The feature score $f_{Spike}(T)$ is then transmitted to the normalizer. When a complete score buffer S_{Spike} becomes available, it is normalized $F_{Spike} = \mathcal{N}_{min-max}(S_{Spike})$ and output in a burst of size w. The normalized Dropout feature vector can be derived from the normalized Spike feature vector: $F_{Dropout} = \mathbf{1}_w - F_{Spike}$.

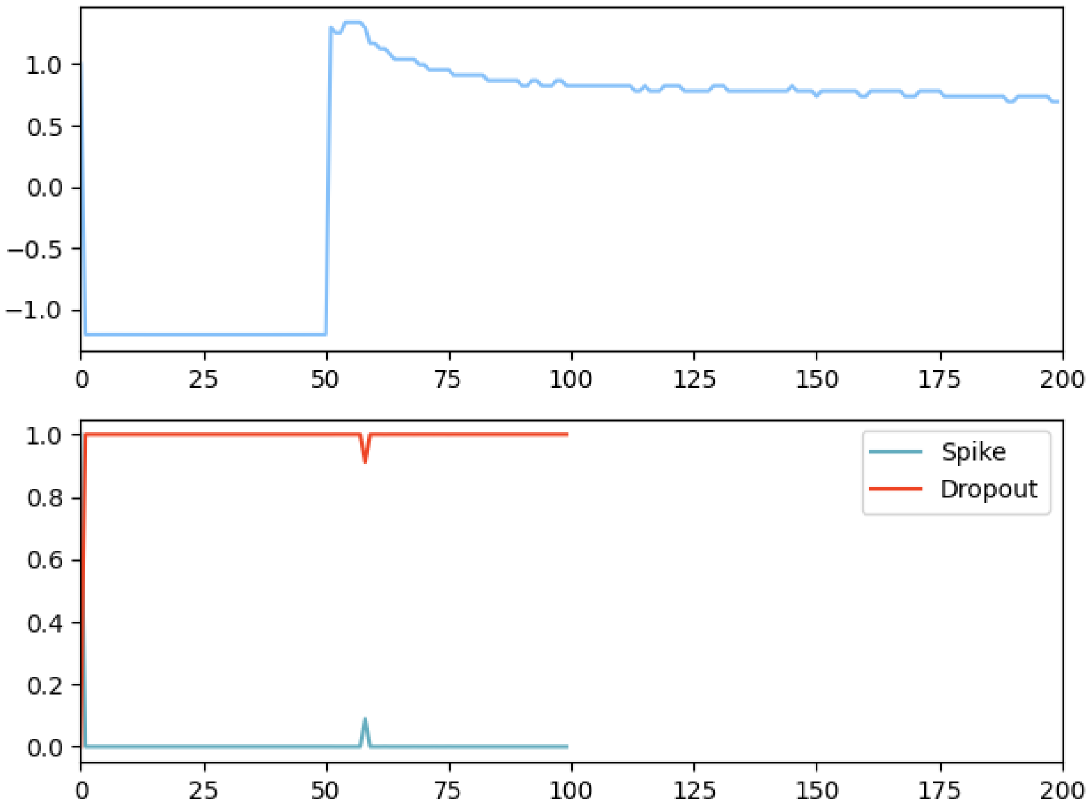


Figure 4.4: **Spike and Dropout:** An example of the Spike and Dropout features computed for the Freezer dataset using a window of size $w = 100$; the Spike and Dropout features are not computed for the last 100 datapoints.

4.5 Rising and Falling

The **Rising** and **Falling** features compute the sum of the slopes between neighboring datapoints in the current window T; Figure 4.5 shows an example.

A function $Slopes : \mathbb{R}^w \to \mathbb{R}^{w-1}$ computes a vector P of length w, where $p_i = t_{i+1} - t_i$. In an offline context, the sum of slopes is the L^1 norm $\|P\|_1 = \|Slopes(T)\|_1$; the streaming feature extractor maintains the sum as a scalar Σ_{Slopes}, which is updated as follows when each new datapoint is sampled:

$$\Sigma_{Slopes} = \Sigma_{Slopes} - (t_2 - t_1) + (t_{w+1} - t_w). \tag{4.5}$$

The feature score $f_{Rising}(T) = \Sigma_{Slopes}$ is transmitted to the normalizer. When a complete score buffer S_{Rising} becomes available, it is then normalized $F_{Rising} = \mathcal{N}^{max-scaled}_{min-max}(F_{Rising})$ and output in a burst of size w. The normalized Falling feature vector can be derived from the normalized Rising feature vector: $F_{Falling} = \mathbf{1}_w - F_{Rising}$.

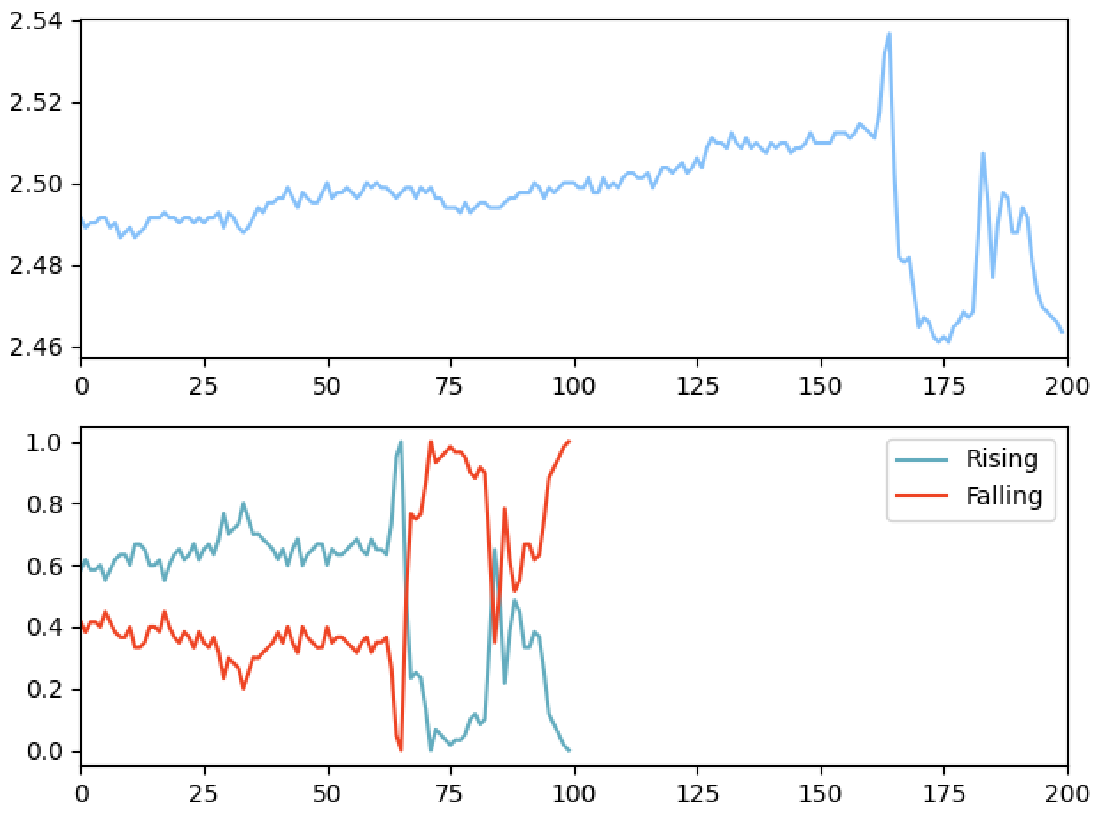


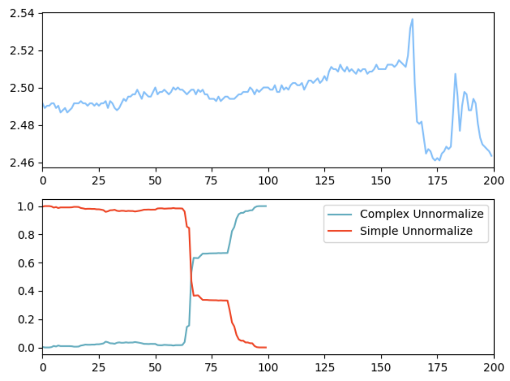
Figure 4.5: **Rising and Falling:** An example of the Rising and Falling features computed for the Insect EPG dataset using a window of size $w = 100$; the Rising and Falling features are not computed for the last 100 datapoints.

4.6 Complex and Simple Unnormalized (Normalized)

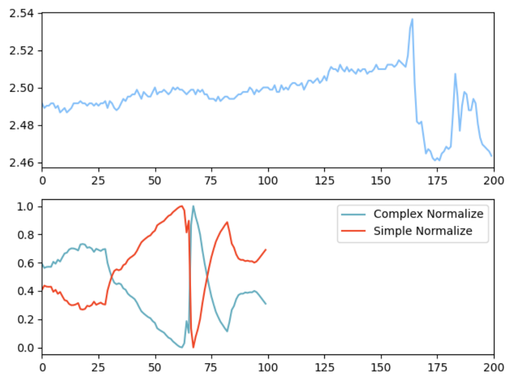
These four features comprise two feature pairs that quantify the intrinsic dimensionality [1] of the current window T. The **Complex (Simple) Unnormalized** features are similar to Rising (Falling), but employ the L^2 norm rather than the L^1 norm of $Slopes(T)$; the **Complex (Simple) Normalized** features are similar to their Unnormalized counterparts, but z-score normalize T before computing the L^2 norm of $Slopes(T)$. Figure 4.6 shows examples for both feature pairs.

The feature score $f_{Complex-Unnormalized}(T) = \|Slopes(T)\|_2$ is transmitted to the normalizer. When a complete score buffer $S_{Complex-Normalized}$ becomes available, it is then normalized $F_{Complex-Unnormalized} = \mathcal{N}_{min-max}(S_{Complex-Unnormalized})$ and output in a burst of size w. The (normalized) Simple Unnormalized feature vector can be derived from the (normalized) Complex Unnormalized feature vector: $F_{Simple-Unnormalized} = \mathbf{1}_w - F_{Complex-Unnormalized}$.

The feature score $f_{Complex-Normalized}(T) = \|Slopes(\mathcal{N}_z(T))\|_2$ is transmitted the normalizer. When a complete score buffer $S_{Complex-Normalized}$ becomes available, it is normalized $F_{Complex-Normalized} = \mathcal{N}_{min-max}(S_{Complex-Normalized})$ and output in a burst of size w. The (normalized) Simple Normalized feature vector can be derived from the (normalized) Complex Normalized feature vector: $F_{Simple-Normalized} = \mathbf{1}_w - F_{Complex-Normalized}$.



(a) Simple and Complex Unnormalized Example



(b) Simple and Complex Normalized Example

Figure 4.6: **Complex and Simple Unnormalized and Normalized:** An example of the Complex Unnormalized, Simple Unnormalized, Complex Normalized, and Simple Normalized features computed for the Insect EPG dataset using a window of size $w = 100$; the features are not computed for the last 100 datapoints.

4.7 Symmetric and Asymmetric

The **Symmetric** and **Assymetric** features quantify the extent to which T is symmetric with respect to its midpoint $t_{n/2}$; Figure 4.7 shows an example.

A function $Rev : \mathbb{R}^w \to \mathbb{R}^w$ reverses T. The feature score $f_{Symmetric}(T) = \|T - Rev(T)\|_2$ is transmitted to the normalizer. When a complete score buffer vector $S_{Symmetric}$ becomes available, it is normalized $F_{Symmetric} = \mathcal{N}^{+}_{min-max}(S_{Symmetric})$. The normalized Asymmetric feature vector can be derived from the normalized Symmetric feature vector: $F_{Asymmetric} = \mathbf{1}_w - F_{Symmetric}$.

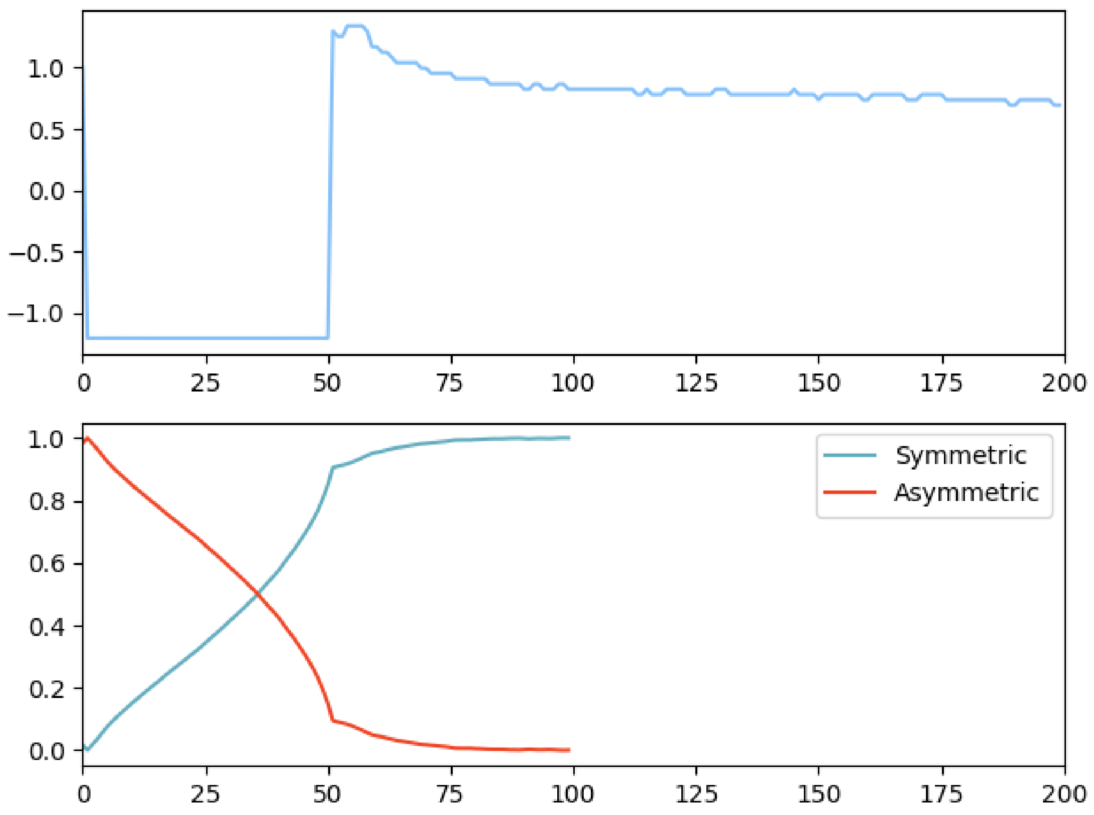


Figure 4.7: **Symmetric and Asymmetric:** An example of the Symmetric and Asymmetric features computed for the Freezer dataset using a window of size $w = 100$; the Symmetric and Asymmetric features are not computed for the last 100 datapoints.

4.8 Linearity and Nonlinearity

The features **Linearity** and **Nonlinearity** quantify how effectively a linear equation $y = ai + b$ can characterize the current window T; Figure 4.8 shows an example. A linear approximation is fit to the datapoints in T, yielding coefficients a and b:

$$b = \frac{\sum_{i=1}^{w} t_i \sum_{i=1}^{w} i^2 - \sum_{i=1}^{w} i \sum_{i=1}^{w} i t_i}{w \sum_{i=1}^{w} i^2 - (\sum_{i=1}^{w} i)^2} \tag{4.6}$$

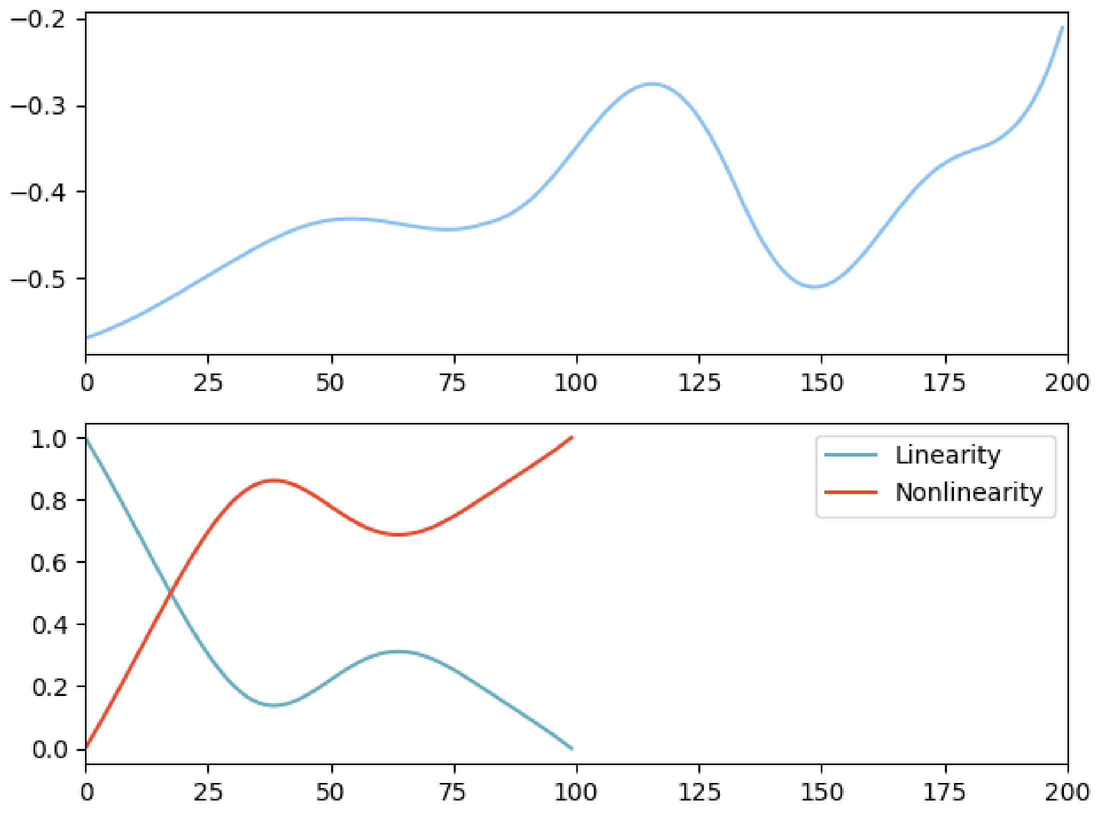


Figure 4.8: **Linearity and Nonlinearity:** An example of the Linearity and Nonlinearity features computed for the Star Light Curve dataset using a window of size $w = 100$; the Linearity and Nonlinearity features are not computed for the last 100 datapoints.

$$a = \frac{w \sum_{i=1}^{w} t_i \sum_{i=1}^{w} i^2 - \sum_{i=1}^{w} i \sum_{i=1}^{w} it_i}{w \sum_{i=1}^{w} i^2 - (\sum_{i=1}^{w} i)^2} \tag{4.7}$$

A function $Linear\_Approx : \mathbb{R}^w \to \mathbb{R}^w$ computes a vector L of length w, where $l_i = ai + b$. The terms a and b are amenable to online updates, as long as we interpret the window size w as a constant; this interpretation is reasonable, as we would need to reconfigure the FPGA to adjust w, and, adjustments to w can change the values produced by each feature extractor. Based on these assumptions, the sums $\sum_{i=1}^{w} i$, $\sum_{i=1}^{w} i^2$, $(\sum_{i=1}^{w} i)^2$, and denominators of both coefficient equations are also constants, and $\sum_{i=1}^{w} t_i$ is the online summation of datapoint values in T computed online by Eq. 4.1.

The sum $\sum_{i=1}^{w} it_i$ can also be computed online. After sampling datapoint t_{w+1}, and prior to discarding t_1 and adjusting datapoint indices, this sum becomes $\sum_{i=1}^{w} it_{i+1}$. Via algebraic manipulation, one can show that $\sum_{i=1}^{w} it_{i+1} = \sum_{i=1}^{w} it_i - \sum_{i=1}^{w} t_i + wt_{w+1}$, once again recognizing that the term $\sum_{i=1}^{w} t_i$ is another instance of Eq. 4.1, which can be computed online.

The Linearity feature is the L^2 norm of the difference between T and its linear approximation. The feature score $f_{Linearity}(T) = \|T - Linear\_Approx(T)\|_2$ is transmitted to the normalizer. When a complete score buffer $S_{Linearity}$ becomes available, it is then normalized $F_{Linearity} = \mathcal{N}^{+}_{min-max}(S_{Linearity})$ and output in a burst of size w. The normalized Nonlinearity feature vector can be derived from the normalized Linearity feature vector: $F_{Nonlinearity} = \mathbf{1}_w - F_{Linearity}$.

4.9 Convex and Concave

The features **Convex** and **Concave** quantify how effectively a quadratic equation $y = ai^2 + bi + c$ can characterize the current window T; Figure 4.9 shows an example. A quadratic approximation is fit to the datapoints in T by solving a system of equations

$M\mathbf{a} = \mathbf{b}$ where,

$$M = \begin{bmatrix} w & \sum_{i=1}^{w} i & \sum_{i=1}^{w} i^2 \\ \sum_{i=1}^{w} i & \sum_{i=1}^{w} i^2 & \sum_{i=1}^{w} i^3 \\ \sum_{i=1}^{w} i^2 & \sum_{i=1}^{w} i^3 & \sum_{i=1}^{w} i^4 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \sum_{i=1}^{w} t_i \\ \sum_{i=1}^{w} it_i \\ \sum_{i=1}^{w} i^2 t_i \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} c \\ b \\ a \end{bmatrix} \tag{4.8}$$

A function $det : \mathbb{R}^{3x3} \to \mathbb{R}$ computes the determinant of a 3x3 matrix. We then solve for $\boldsymbol{a}$ by using Cramer's Rule, where k is the index of vector $\boldsymbol{a}$, and M_k denotes M with the k^{th} column replaced by vector $\boldsymbol{b}$:

$$a = \frac{det(M_3)}{det(M)} \quad b = \frac{det(M_2)}{det(M)} \quad c = \frac{det(M_1)}{det(M)} \tag{4.9}$$

Similar to our discussion of Linearity/Nonlinearity above, matrix M contains nine scalar constants (under the assumption that w is constant), making the determinant $det(M)$ also a constant. The sums in vector $\boldsymbol{b}$ can be computed online analogously to Eq. 4.1.

A function $Quadratic\_Approx : \mathbb{R}^w \to \mathbb{R}^w$ computes a vector Q of length w, where $q_i = ai^2 + bi + c$. The Concave feature is the L^2 norm of the difference between T and its quadratic approximation. The feature score $f_{Convex}(T) = \|T - Quadratic\_Approx(T)\|_2$ is transmitted to the normalizer. When a complete score buffer S_{Convex} becomes available, it is then normalized $F_{Convex} = \mathcal{N}^{+}_{min-max}(S_{Convex})$

and output in a burst of size w. The normalized Concave feature vector can be derived from the normalized Convex feature vector: $F_{Concave} = \mathbf{1}_w - F_{Convex}$.

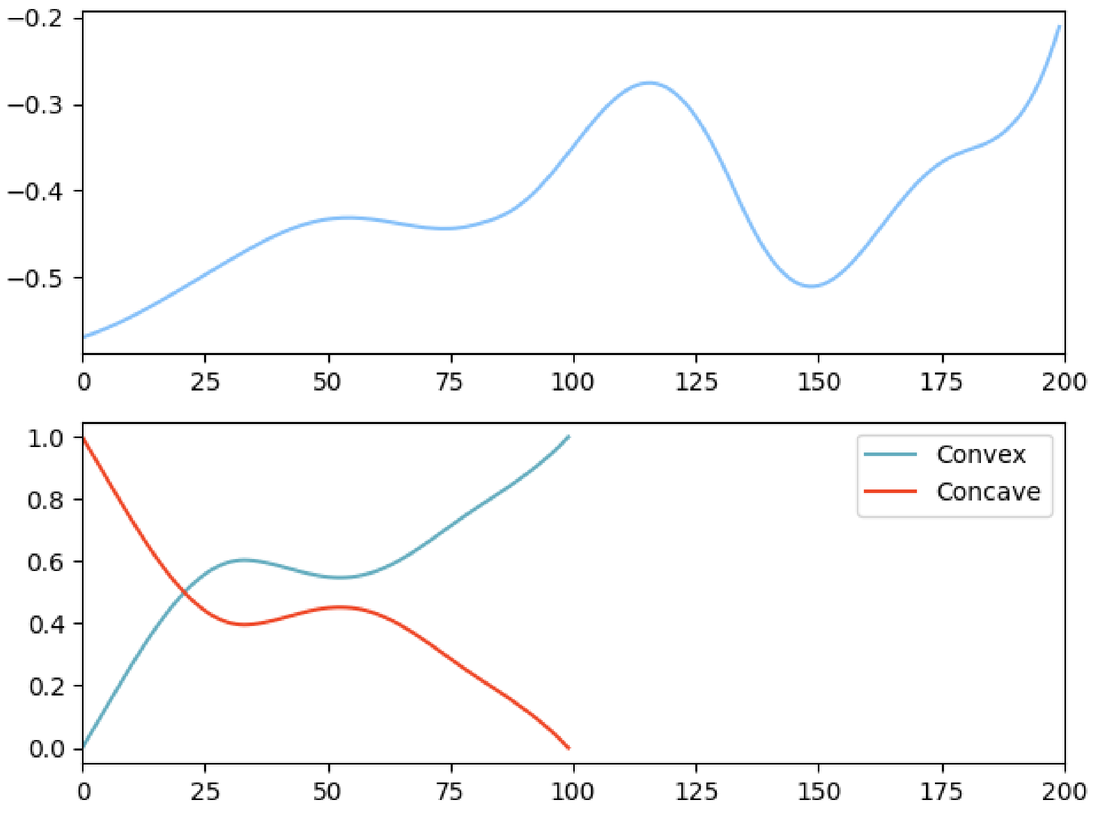


Figure 4.9: **Convex and Concave:** An example of the Convex and Concave features computed for the Star Light Curve dataset using a window of size $w = 100$; the Convex and Concave features are not computed for the last 100 datapoints.

4.10 Noise and Smooth

The **Noise** and **Smooth** features estimate the amount of noise in the current window T by computing the sum of the root squared difference of the original window and a smoothed version; Figure 4.10 shows an example. These features are computed by composing three functions introduced next.

Function $Smooth : \mathbb{R}^w \rightarrow \mathbb{R}^w$ computes a smoothed version of the window using a median filter with $k = 2$, i.e., $Smooth(T) = MedianFilter(T, 2)$.

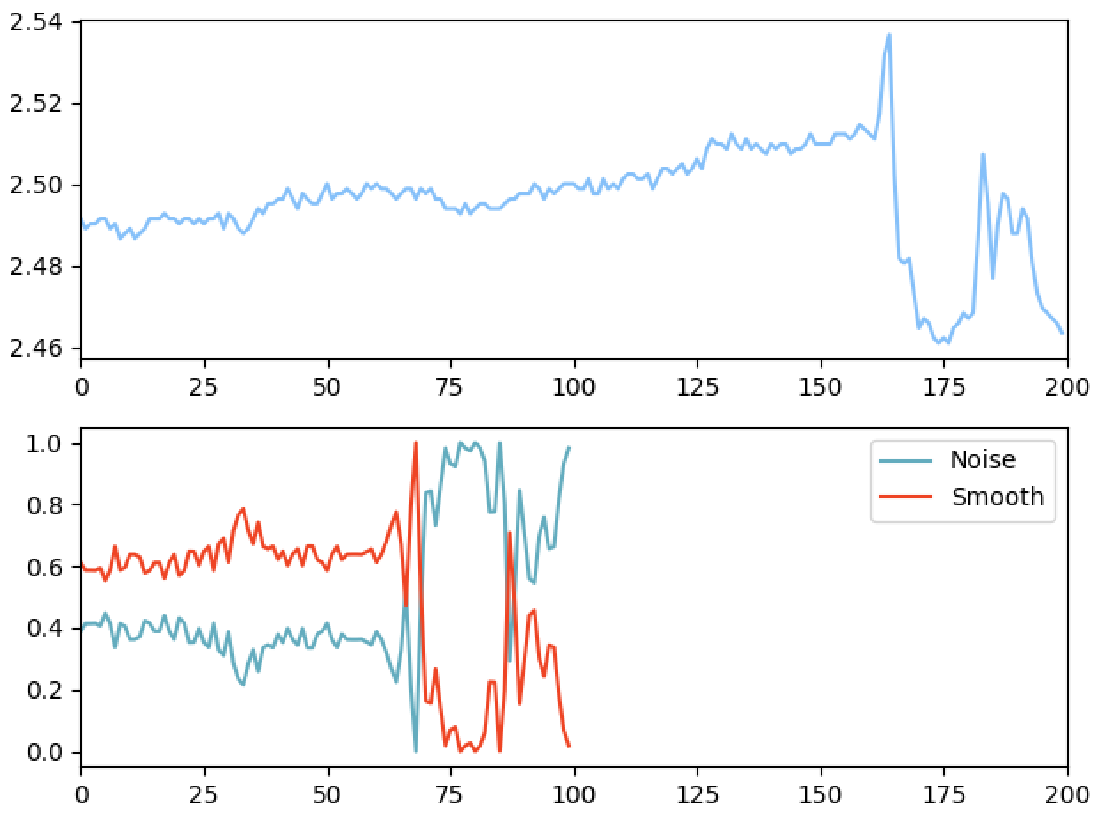


Figure 4.10: **Noise and Smooth:** An example of the Noise and Smooth features computed for the Insect EPG dataset using a window of size $w = 100$; the Noise and Smooth features are not computed for the last 100 datapoints.

Function $Diff : \mathbb{R}^w \times \mathbb{R}^w \rightarrow \mathbb{R}^w$ computes the elementwise absolute value of the difference of two vectors T and U: $d_i = |t_i - u_i|$. Computing the absolute difference between each sampled datapoint and its smoothed counterpart serves as a proxy for noise.

Function $Remove\_Outliers : \mathbb{R}^w \to \mathbb{R}^w$ removes outliers from a vector T. Let μ and σ be the arithmetic mean and standard deviation of the datapoints in T. Datapoint t_i is an inlier if $\mu - \sigma \leq t_i \leq \mu + \sigma$; otherwise, t_i is an outlier and is replaced by the nearest datapoint (by index) in T that is also an inlier, which we denote as in_i. $Remove\_Outliers$ stores its result in a vector R of length w, where:

$$r_i = \begin{cases} t_i & \mu - \sigma \leq t_i \leq \mu + \sigma \\ in_i & \text{otherwise} \end{cases} \tag{4.10}$$

The Noise feature is the L^1 norm of the difference between T and its smoothed counterpart, after removing outlier points.

The feature score $f_{Noise}(T) = \|Remove\_Outliers(Diff(T, Smooth(T))\|_1$ is transmitted to the normalizer. When a complete score buffer S_{Noise} becomes available, it is then normalized $F_{Noise} = \mathcal{N}_{min-max}(S_{Noise})$ and output in a burst of size w. The normalized Smooth feature vector can be derived from the normalized Noise feature vector: $F_{Smooth} = \mathbf{1}_w - F_{Noise}$.

4.11 Step and No Step

The **Step** and **No Step** features quantify how well the current window T can be characterized by a step-wise function. Figure 4.11 shows an example. In addition to z-score normalization ($\mathcal{N}_z$) and the *Remove_Outliers* functions, introduced earlier, three additional functions are introduced here:

Function $Zero\_Min : \mathbb{R}^w \to \mathbb{R}^w$ vertically shifts a time series T so that the smallest datapoint has a value of zero. Let t_{min} be the minimum value among all datapoints in T. Then $Zero\_Min(T) = T - t_{min} \cdot \mathbf{1}_w$.

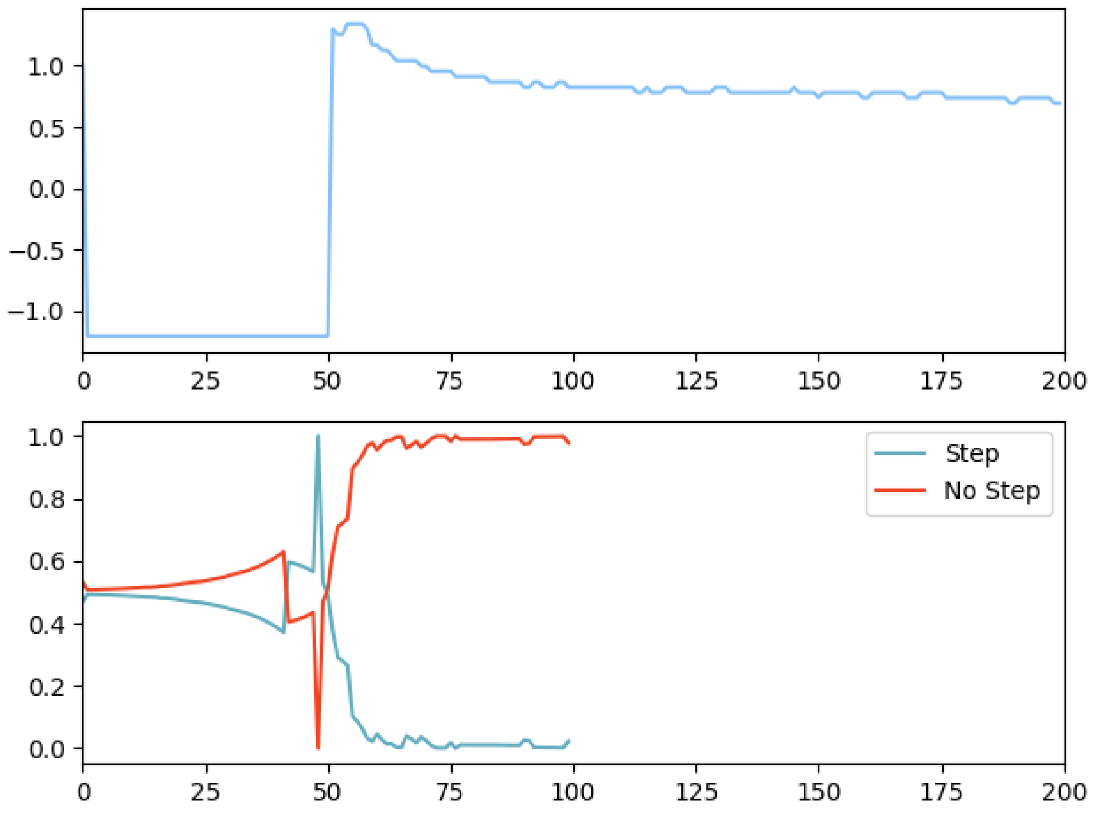


Figure 4.11: **Step and No Step:** An example of the Step and No Step features computed for the Insect EPG dataset using a window of size $w = 100$; the Step and No Step features are not computed for the last 100 datapoints.

Function *Rate_of_Change* : $\mathbb{R}^w \to \mathbb{R}^w$ computes the rate of change between neighboring datapoints of T, storing the result in a vector $\Delta = \langle \delta_1, \delta_2, \ldots, \delta_w \rangle$, where

$$\delta_i = \begin{cases} t_{i+1} - t_i & 1 \leq i \leq w - 1 \\ 0 & i = w \end{cases} \tag{4.11}$$

Function *Stepwise* : $\mathbb{R}^w \to \mathbb{R}^w$ estimates a step function that reasonably characterizes time series T. The first step is to identify the indices of outlier datapoints in

T which satisfy the property $t_i > 3\sigma$, where σ is the standard deviation of T. These indices are collected into a vector $K = \langle i|\ \delta_i > 3\sigma \wedge i-1 \notin K\rangle$. By convention, we require K to contain index 1, and K is not permitted to contain consecutive indices. The length of K, denoted m, is data dependent and satisfies $m \leq w/2$.

The *Stepwise* function computes a vector E using the indices in K. For index i, let k_l and k_{l+1} be the entries of K that satisfy $k_l \leq i < k_{l+1}$. Then the estimated step e_i at position i is computed as follows:

$$e_i = \frac{\sum_{j=k_l}^{k_{l+1}} t_j}{k_{l+1} - k_l} \quad k_l \leq i < k_{l+1} \tag{4.12}$$

There is a subtle discontinuity involving the final entry of K. There are two cases to consider:

1. If $w \in K$, i.e., if the last entry of K, $k_m = w$, then we must compute e_w for $k_m = w < k_{m+1}$, where k_{m+1} is undefined because it exceeds the length of K (i.e., an out-of-bounds index). In this case, we simply let $e_w = t_w$.

2. If $w \notin K$, then $k_m = v < w$, and k_{m+1} is likewise undefined. In this case, we use Eq. 4.12 with $k_l = v$ and $k_{l+1} = w$, essentially treating the trailing indices of T as the final step.

To compute the Step feature, let

$$A = Zero\_Min(Remove\_Outliers(\mathcal{N}_z(T))) \tag{4.13}$$

and

$$B = Stepwise(Rate\_of\_Change(A)). \tag{4.14}$$

The Step feature is the L^2 norm of the difference between A (the current window, after z-score normalization, outlier removal, etc.) and B (the stepwise estimation). The feature score $f_{Step}(T) = \|A - B\|_2$ is transmitted to the normalizer. When a complete score buffer S_{Step} becomes available, it is then normalized $F_{Step} = \mathcal{N}^{+}_{min-max}(S_{Step})$ and output in a burst of size w. The normalized No Step feature vector can be derived from the normalized Step feature vector: $F_{No-Step} = \mathbf{1}_w - F_{Step}$.

4.12 Periodic and Aperiodic

The **Periodic** and **Aperiodic** features quantify the ability of two Fast Fourier Transform (FFT) coefficients to model the window of recently sampled datapoints. Figure 4.12 shows an example. This function employs z-score normalization as a preliminary step, which is then followed by the following functions.

Functions $FFT : \mathbb{R}^w \to \mathbb{R}^w$ and $IFFT : \mathbb{R}^w \to \mathbb{R}^w$ transform the datapoints between the time and frequency domains.

Function $Subst\_Max_4 : \mathbb{R}^w \to \mathbb{R}^w$ identifies and extracts the 4-largest values from a vector of length $w \geq 4$, places them in-order in the first four positions, and zeroes out the remaining value. For example, let $v^1_{max} \geq v^2_{max} \geq v^3_{max} \geq v^4_{max}$ be the four largest values in vector V. Then $Subst\_Max_4(V) = \langle v^1_{max}, v^2_{max}, v^3_{max}, v^4_{max}, 0, 0, \ldots, 0\rangle$.

Let $Y = IFFT(Subst\_Max_4(FFT(\mathcal{N}_z(T))))$. To compute Y, the current window T is z-score normalized and then transformed into the frequency domain via FFT. Within the frequency domain, the four highest-valued frequency domain coefficients are identified and moved to the front of the vector, while zeroing out all other coefficients, before transforming it back into the time domain via IFFT. The Periodic feature is the L^2 norm of the difference between Y and the z-score normalized current window T.

The frequency score $f_{Periodic}(T) = \|\mathcal{N}_z(T) - Y\|_2$ is transmitted to the normalizer. When a complete score buffer $S_{Periodic}$ becomes available, it is then normalized $F_{Periodic} = \mathcal{N}_{min-max}(S_{Periodic})$ and output in a burst of size w. The normalized Aperiodic feature vector can be derived from the normalized Periodic feature vector $F_{Aperiodic} = \mathbf{1}_w - F_{Periodic}$.

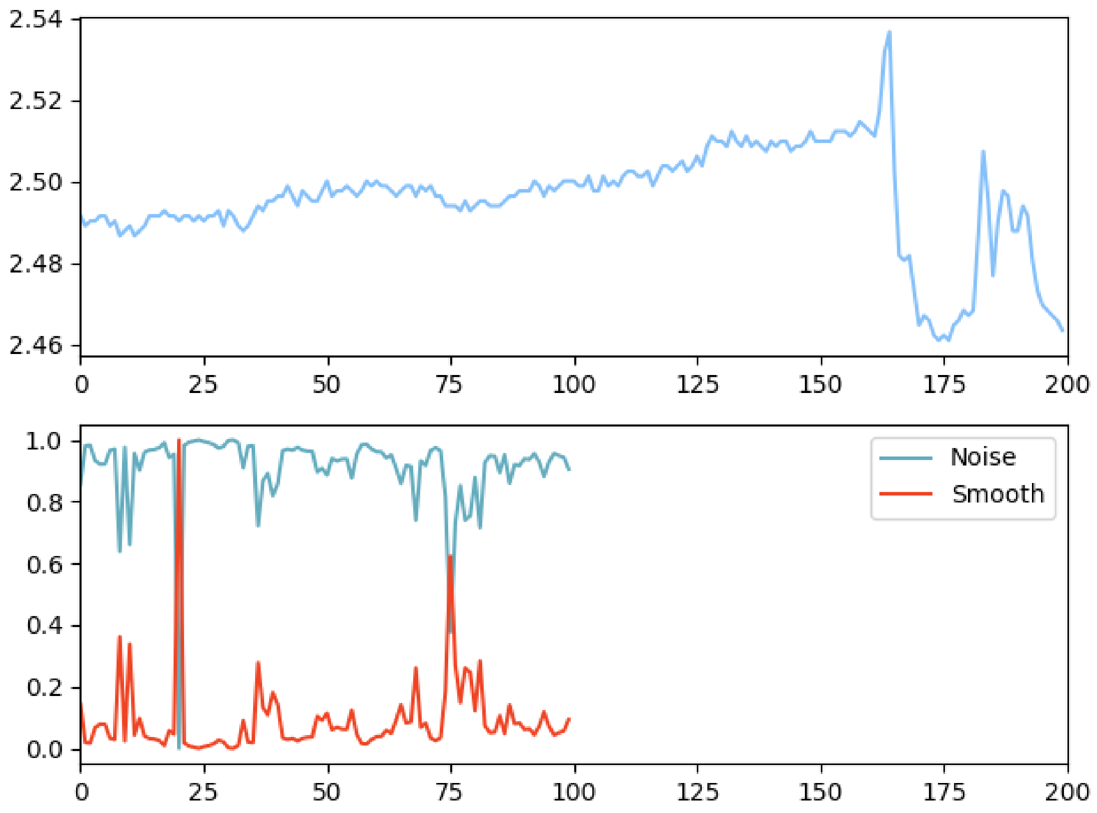


Figure 4.12: **Periodic and Aperiodic:** An example of the Periodic and Aperiodic features computed for the Insect EPG dataset using a window of size $w = 100$; the Periodic and Aperiodic features are not computed for the last 100 datapoints.

4.13 Unique Pattern

The **Unique Pattern** feature calculates the amount of unique values within the window of recently sampled datapoints. Figure 4.13 shows an example. This feature extractor is implemented by a function $Unique : \mathbb{R}^w \rightarrow \mathbb{R}$ defined as

$$Unique(T) = |\{t_i \in T |\ t_j \neq t_i\ \forall t_j \in T \setminus \{t_i\}\}|. \tag{4.15}$$

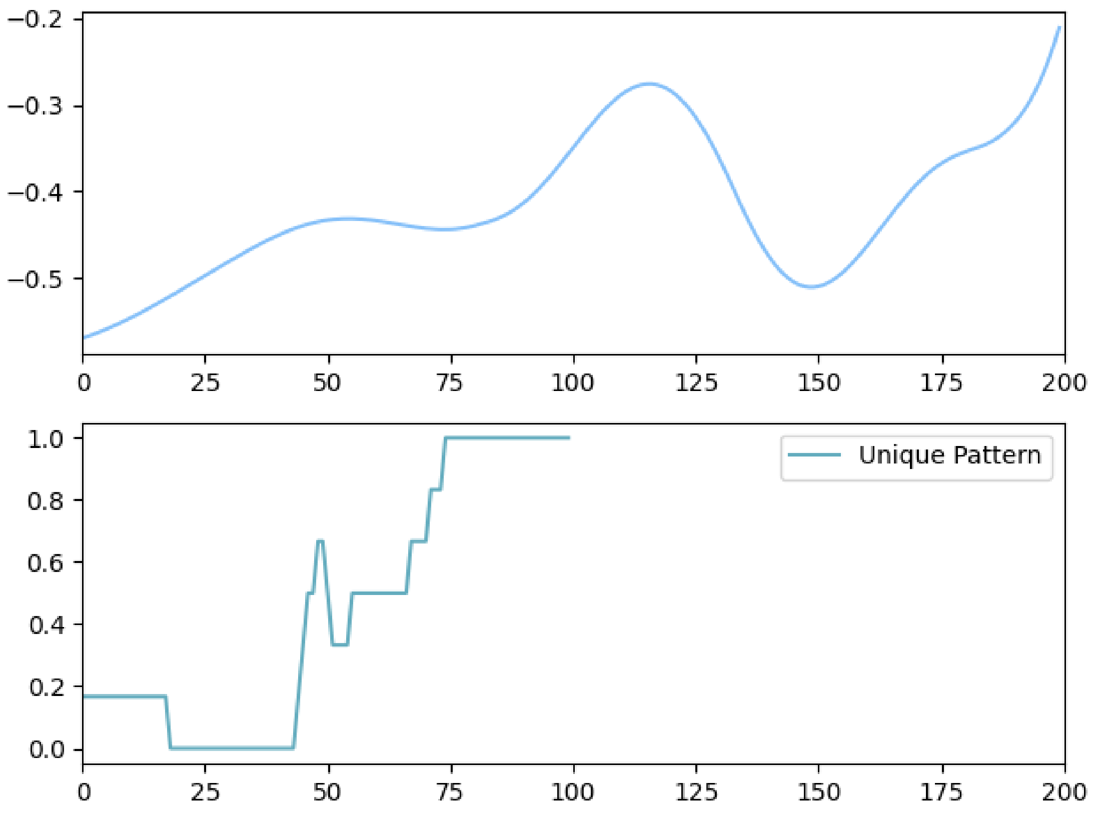


Figure 4.13: **Unique Patterns:** An example of the Unique Patterns feature computed for the Star Light Curve dataset using a window of size $w = 100$; the Unique Patterns feature is not computed for the last 100 datapoints.

The feature score $f_{Unique}(T) = Unique(T)$ is transmitted to the normalizer. When a complete score buffer S_{Unique} becomes available, it is then normalized $F_{Unique} = \mathcal{N}_{min-max}(S_{Unique})$ and output in a burst of size w.

Chapter 5

FPGA Implementation

This chapter presents an FPGA accelerator for Streaming NLP Time Series Search. The accelerator extracts 25 features comprising eleven feature pairs and three standalone features (Table 4.1). Figure 5.1 presents the accelerator: the FPGA obtains data in real time from a sensor (possibly over a network), extracts the 25 features and transmits them to a host PC for storage and/or subsequent processing. The **Feature Extraction Kernel** samples the time series datapoints from the sensor, and transmits each datapoint via FIFOs to eight parallel regions that concurrently compute 14 feature scores and write them to their respective score buffers (BRAM). When the score buffers are full (i.e., when their capacity reaches w), they are normalized (in accordance with Table 4.1) and written to the host (via FIFO) as feature vectors; 11 of the 14 feature scores are concurrently inverted to generate their respective feature pairs, are are also written to the host (once again, via FIFO). Each burst generates a total of $25 \times w$ scalar values to be written to the host.

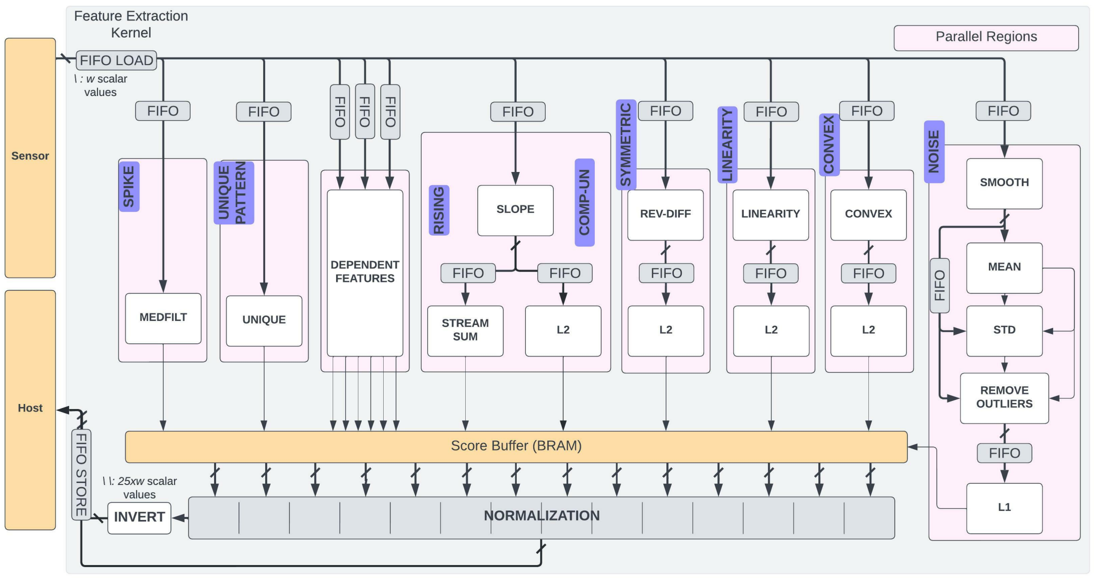


Figure 5.1: **FPGA Accelerator Architecture:** A time series streams into the FPGA from a sensor and is distributed to 8 parallel regions that compute 14 feature scores, which are buffered and normalized to yield 14 feature vectors; 11 of the 14 feature vectors are inverted to generate paired feature vectors. The accelerator outputs 25 "meta-time series" feature vectors to the host for storage or subsequent processing.

Six of the eight parallel regions compute individual feature scores (**Spike**, **Unique Pattern**, **Symmetric**, **Linearity**, **Convex**, **Noise**); a seventh parallel region computes two feature scores (**Rising**, **Complex-Unnormalized**), which share a hardware module that computes $Slopes(T)$ (see Section 4.5); the eighth parallel region, denoted **Dependent Features**, computes six features (**Low**, **High**, **High Amplitude**, **Complex-Normalized**, **Periodic**), with a significant number of shared hardware modules (see Figure 5.2).

The presentation of hardware modules in each parallel region mimics the presentation of each feature scoring method in the preceding chapter. Each parallel region is pipelined internally, and FIFOs are used to communicate array data between hardware modules. The following subsections describe the hardware implementation in greater detail.

5.1 Dependent Region

The Dependent Region (see Figure 5.2) computes six features, whose dependency tree is shown in Figure 5.3. The feature extraction kernel (also shown in Figure 5.1), generates the current time series window T. The streaming sum (of the elements of T; Eq. (4.1)) computed for **Low** is used to compute the streaming mean μ (Eq. (4.2)) for **High**, which is used to compute the standard deviation σ (Eq. (4.3)) for **High Amplitude**; μ and σ are then used to z-score normalize T (Eq. (3.3)). **Complex Normalize**, **Periodic**, and **Step** subsequently process T after z-score normalization, and can execute concurrently.

5.2 Arithmetic Modules

This section summarizes the different arithmetic modules (white boxes) shown in Figures 5.1 and 5.2; the Normalization and Invert modules (shown in Figure 5.1)

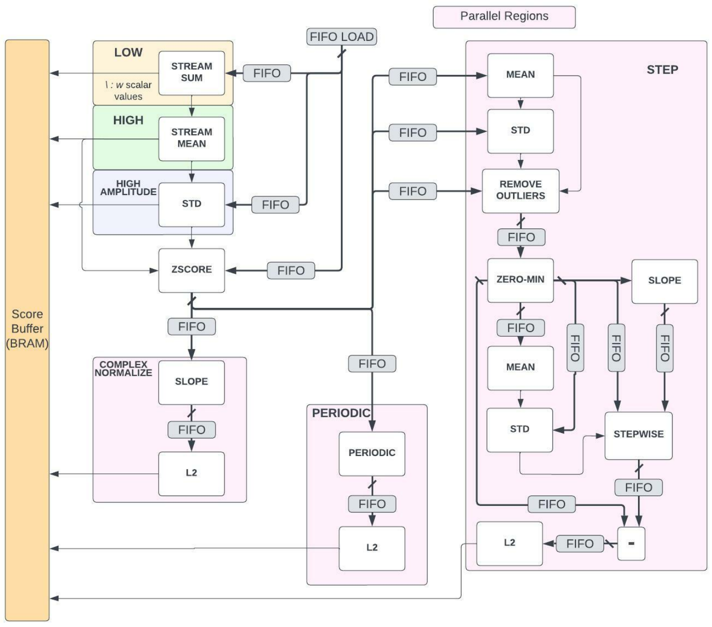


Figure 5.2: **Dependent Region:** The **Low**, **High**, and **High Amplitude** features scores are computed in-order, followed by z-score normalization of the time series window; once the z-score normalized time series window is available, the **Complex Normalize**, **Periodic**, and **Step** features are computed in parallel.

will be discussed later, in Section 5.3. Many of the arithmetic operators shown in this section, such as division or square root, are pipelined; internal registers are not shown.

5.2.1 Streaming Sum and Mean

Figures 5.4a and 5.4b depict the Streaming Sum and Streaming Mean hardware modules in the Dependent Region (Figure 5.2), which implement Eqs. (4.1) and (4.2), where the Streaming Sum is divided by the window size w to compute the Streaming

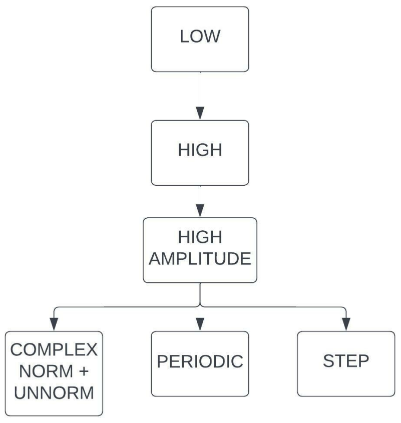
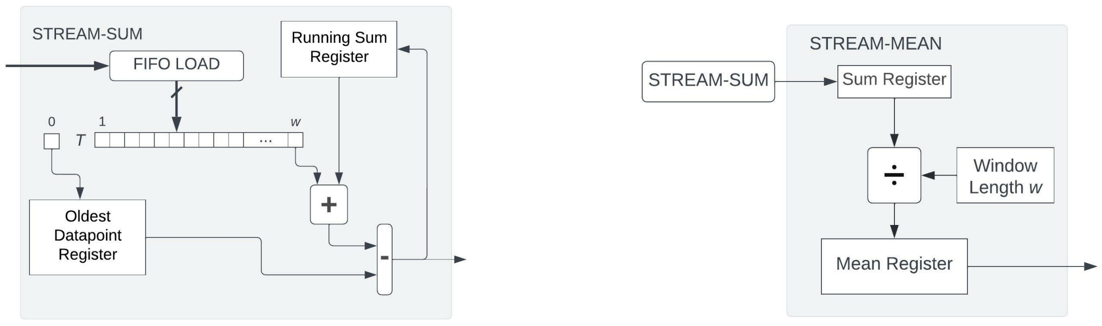


Figure 5.3: Dependency Tree

Mean. The Streaming Sum is also used to compute the **Rising** feature score (Fig. 5.1).

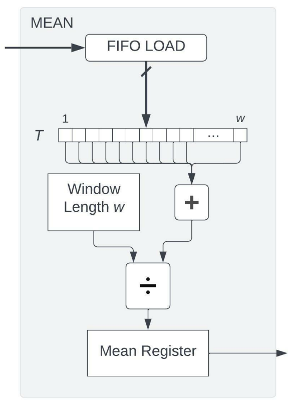


(a) Streaming Sum (b) Streaming Mean

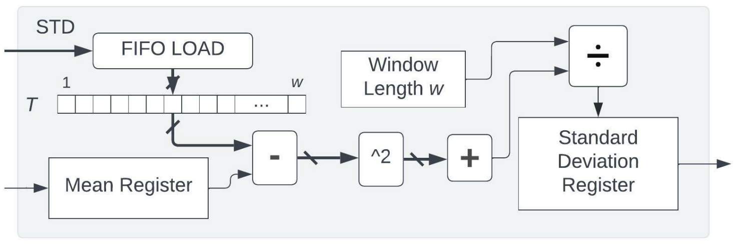
Figure 5.4: Streaming Sum and Streaming Mean hardware modules.

5.2.2 Mean and Standard Deviation (Std)

Figures 5.5a and 5.5b depict a non-streaming mean and standard deviation (STD) (Eq. 4.3) respectively. In both cases, the entire time series is available as a vector, and the mean and standard deviation are computed across all vector entries. These modules are paired in in the parallel regions that compute the **Noise** (Fig. 5.1) and **Step** (Fig. 5.2) feature scores; the **High Amplitude** feature score uses the Streaming Mean computed for the **High** feature score to compute a standard deviation using STD as well (Fig. 5.2).



(a) Mean



(b) Standard Deviation (STD)

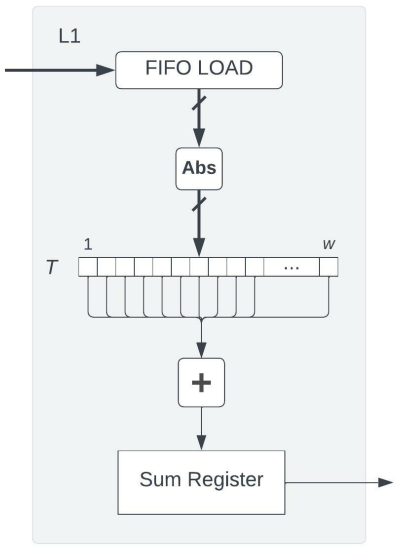
Figure 5.5: (Non-streaming) Mean and Standard Deviation (STD) hardware modules.

5.2.3 L^1 and L^2 Norms

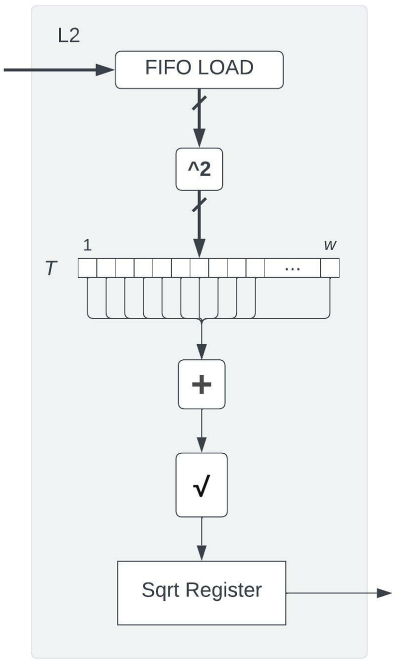
Figures 5.6a and 5.6b depict hardware modules that compute the L^1 and L^2 norms (Eqs. (3.1) and 3.2)). Across Figs. 5.1 and 5.2, the L^1 norm is computed once (for the **Noise** feature score), while seven feature scores compute L^2 norms internally.

5.2.4 z-Score Normalization

Figure 5.7 depicts the hardware module that implements z-score normalization (Eq. 3.3). The mean and standard deviation are pre-computed by other hardware modules. The z-score computation is implemented as a fully-unrollable vector operation, in which each scalar operation performs subtraction followed by division. z-score normalization is computed once in the Dependent Region and is used to compute the **Complex Normalized**, **Periodic**, and **Step** features.



(a) L^1 Norm



(b) L^2 Norm

Figure 5.6: L^1 and L^2 norm hardware modules.

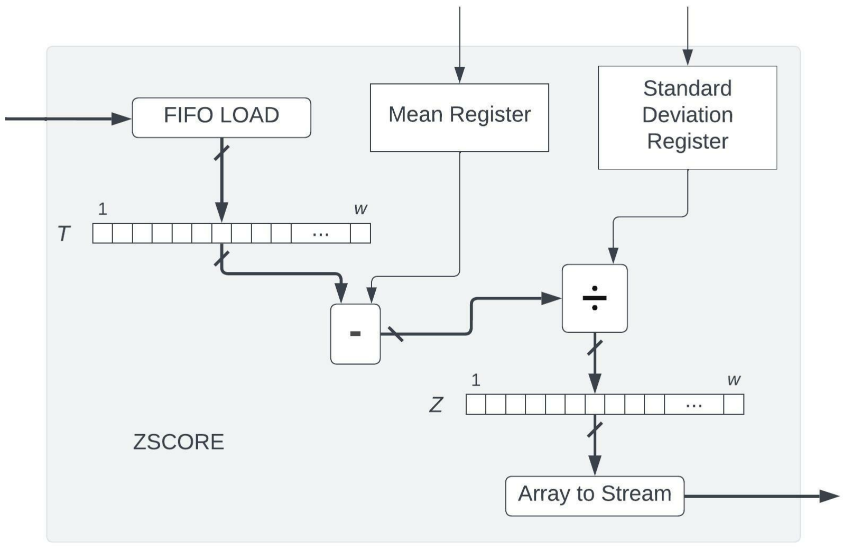


Figure 5.7: z-score Normalization hardware module.

5.2.5 Slope

Figure 5.8 depicts the hardware module that implements the function *Slopes* introduced in Section 4.5, which is a fully unrollable vector operation that computes $p_i = t_{i+1} - t_i, 1 \leq i \leq w - 1$, and sets $p_w = 0$. The Slopes hardware module is used to compute the **Rising**, **Step**, **Complex Normalized**, and **Complex Unnormalized** feature scores in Figures 5.1 and 5.2.

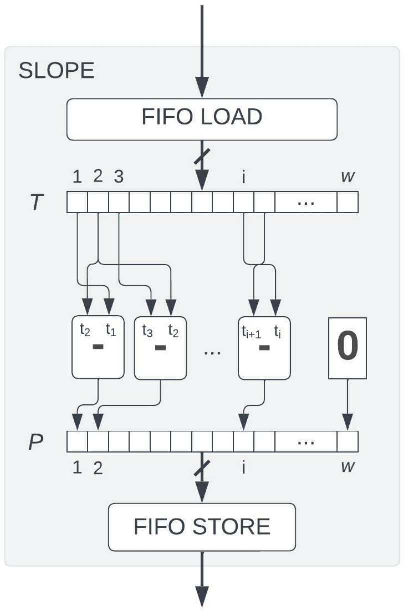
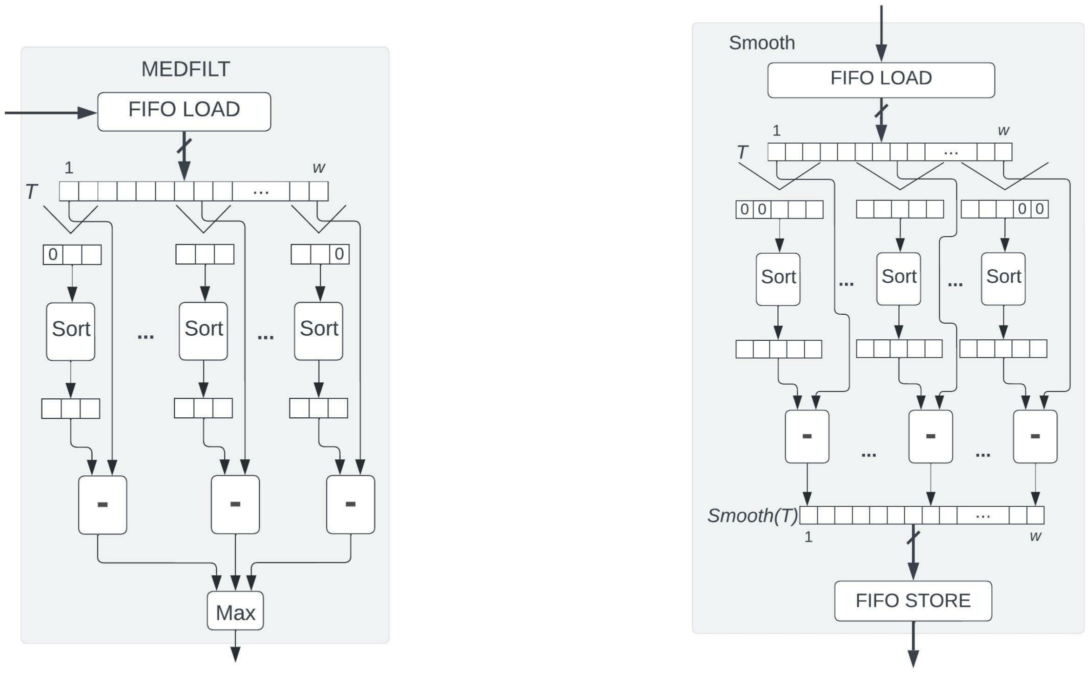


Figure 5.8: Slope hardware module.

5.2.6 Median Filter and Smooth

Figures 5.9a and 5.9b depicts hardware modules that implements the Median Filter and the Smooth function introduced in Sections 4.4 and 4.10. The two functions are quite similar: for each datapoint t_i, they compute the median of the subsequence $\langle t_{i-k}, t_{i-k+1}, \dots t_{i+k} \rangle$ at position i. The Median Filter outputs the maximum of the computed medians, while the Smooth function stores the medians in a vector. As k tends to be small, each median is computed efficiently by sorting, and all medians can be computed as a fully unrollable vector operation. The Median Filter is used to compute the **Spike** feature score, while the Smooth function is used to compute the **Noise** feature score (Figure 5.1).

To facilitate efficient pipelining, median computations that involve indices outside of the range $[1, w]$ are handled separately; these edge cases execute concurrently with the main loop, which is fully unrolled; this approach eliminates the need to pad the vector with additional data points or to insert inefficient branching statements into the loop that specifies the computation.



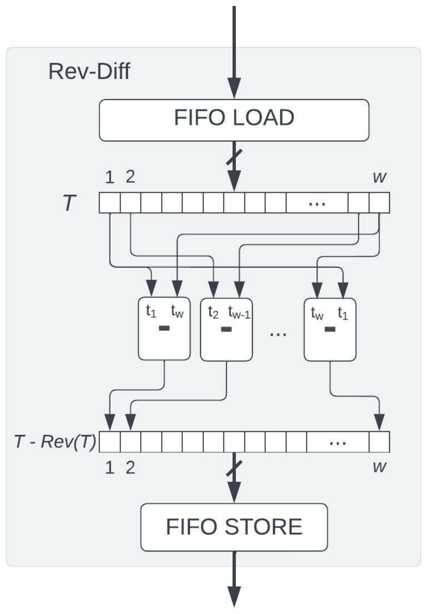
(a) Median Filter (b) Smooth

Figure 5.9: Median Filter and Smooth hardware modules.

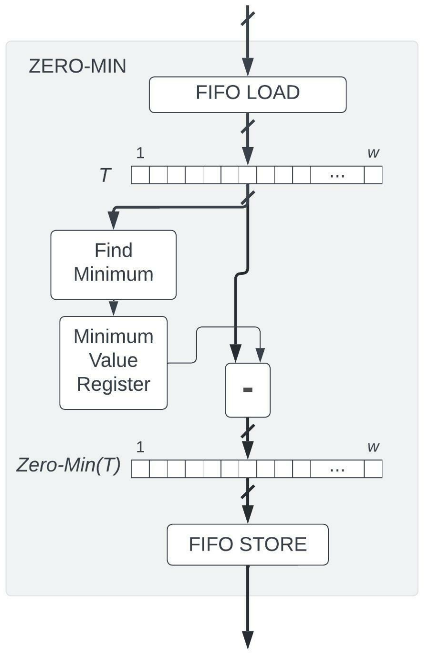
5.2.7 Rev-Diff and Zero_Min

Figure 5.10a depicts a hardware module that computes the function $T - Rev(T)$ described in Section 4.7, which is used to compute the **Symmetric** feature score (Figure 5.1). Figure 5.10b depicts a hardware module that computes the function

Zero_Min described in Section 4.11, which is used to compute the **Step** feature score (Figure 5.2). Both of these modules can be implemented as fully-unrollable vector subtraction operations, with *Zero_Min* first requiring computation of the minimum value of the vector.



(a) Rev-Diff



(b) Zero_Min

Figure 5.10: Rev-Diff and Zero_Min hardware modules.

5.2.8 Remove_Outliers

Figures 5.11 depicts a hardware module that computes the *Fill_Outliers* function introduced in Section 4.10, which is used to compute the **Noise** (Figure 5.1) and **Step** (Figure 5.2) feature scores. The threshold for outlier detection is $\mu \pm \sigma$; an array is created by identifying the inliers (i.e., non-outliers) and replacing each outlier by its nearest inlier.

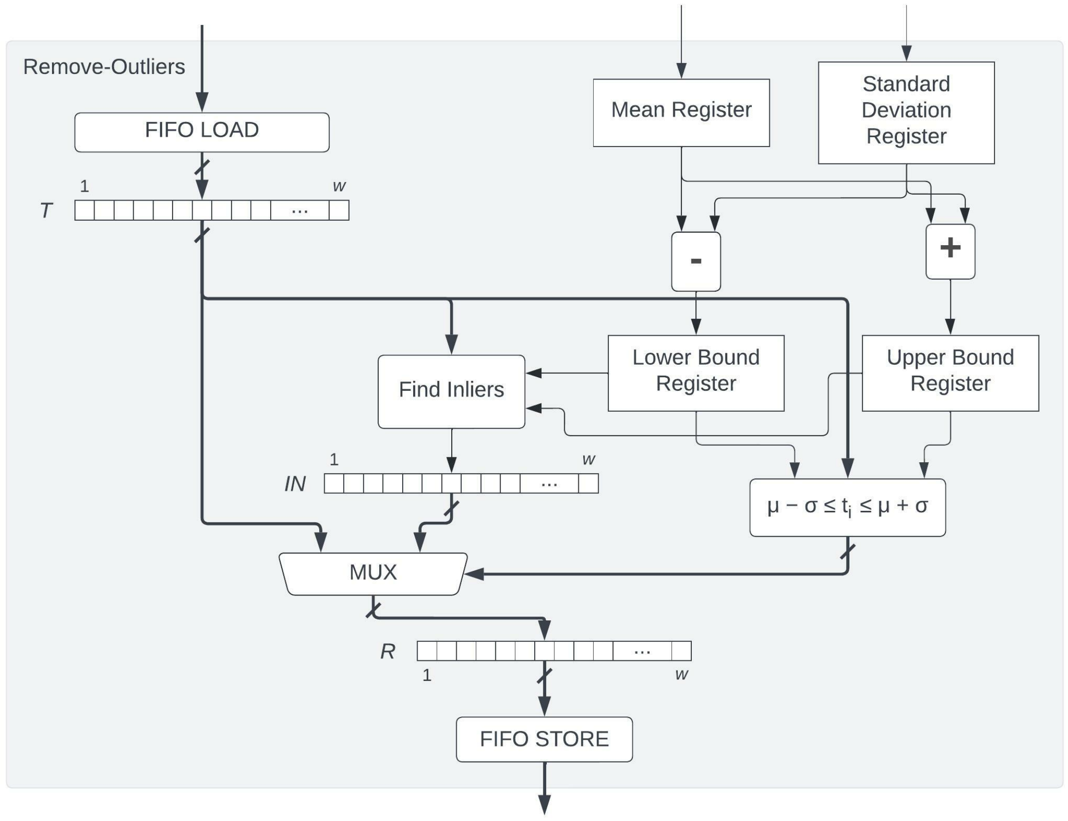


Figure 5.11: Remove_Outliers hardware module.

5.2.9 Stepwise

Figure 5.12 computes the Stepwise function (Eq. (4.14)) described in Section 4.11, which is used to compute the **Step** feature score (Figure 5.2). Internally, the absolute value of the slopes produced by the Slope hardware module (Figure 5.8) is used to identify indices where new steps occur, i.e., where the absolute value of the slopes exceeds 3σ. The stepwise approximateion is created by setting the values of a step to the mean value of the subarray based on the indices of the step.

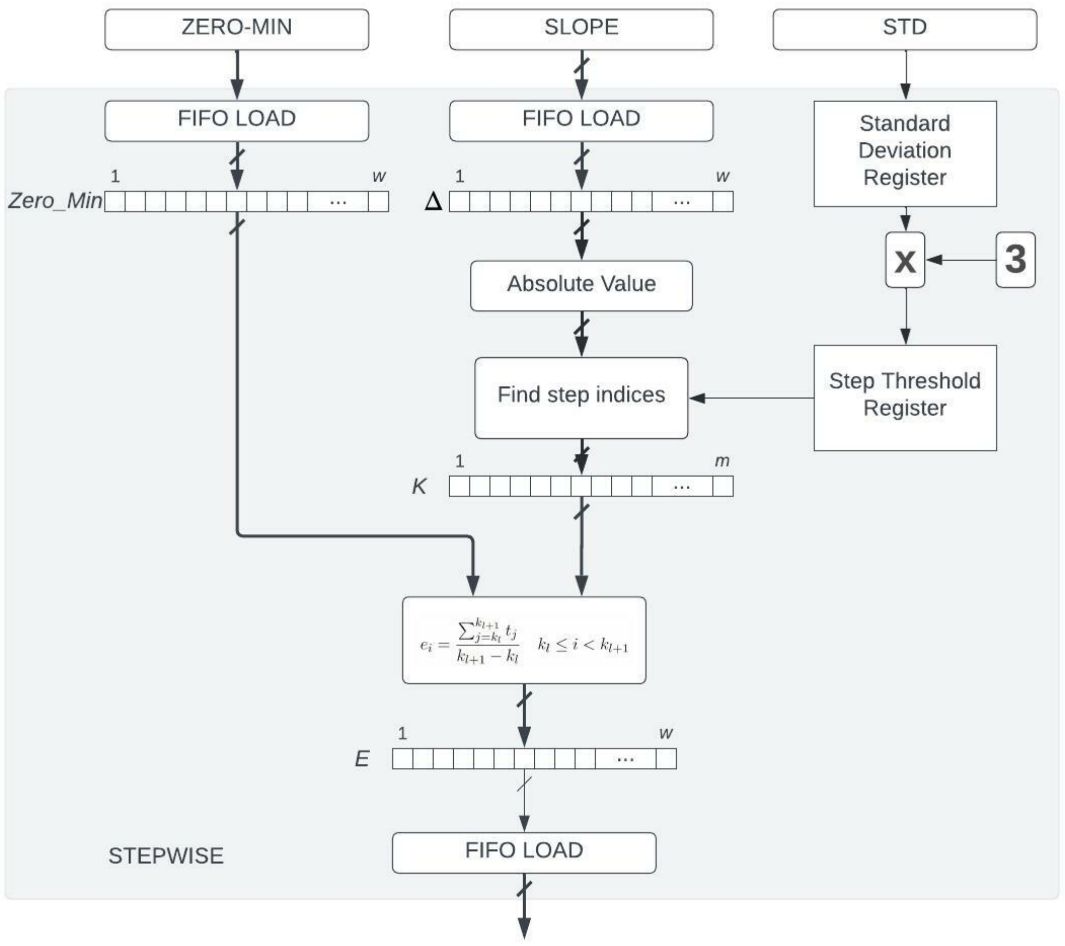
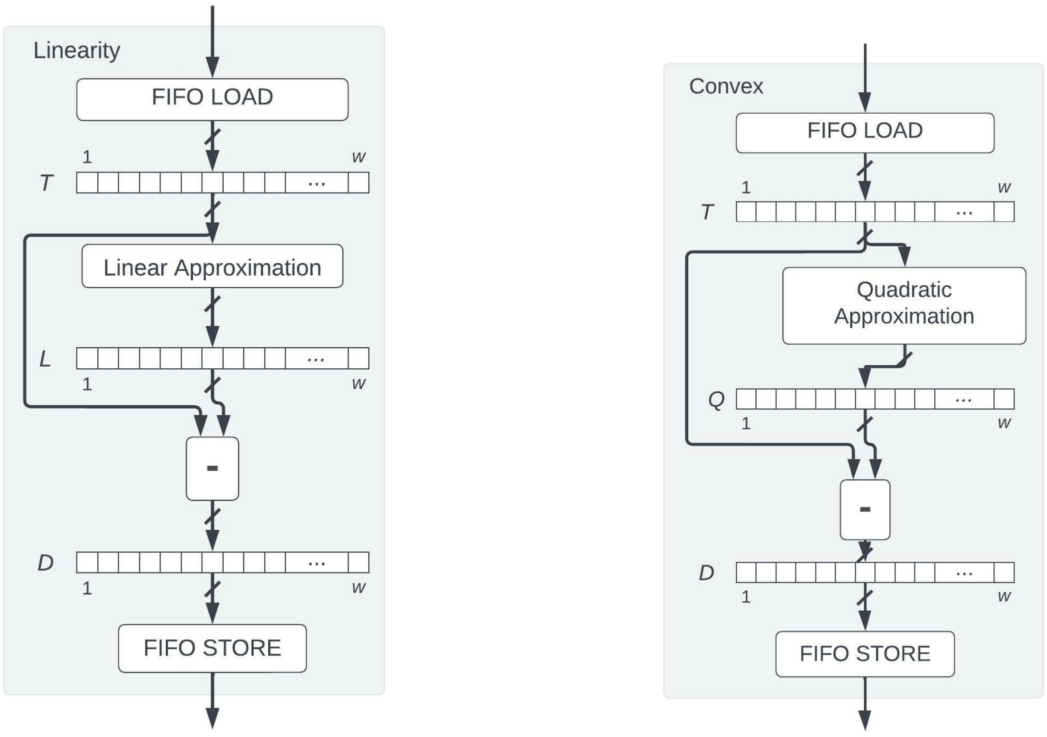


Figure 5.12: Stepwise hardware module.

5.2.10 Linearity and Convex

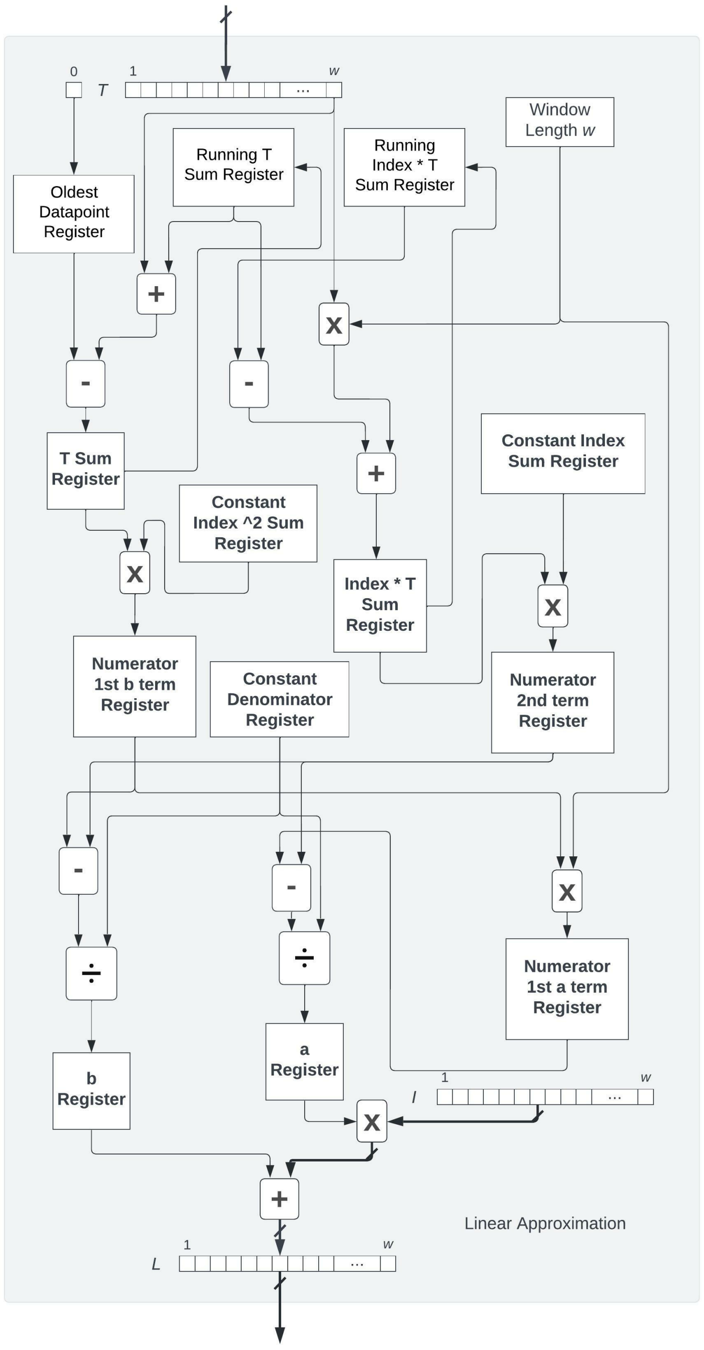
Figures 5.13a and 5.13b depict hardware modules that compute the **Linearity** and **Convex** feature scores (Fig. 5.1). The **Linearity** feature score computes the difference between a time series and a Linear Approximation of that time series, while the **Convex** feature score computes the difference between a time series and a quadratic approximation.

Figure 5.14 depicts the hardware module that computes the linear approximation of a time series via linear regression in accordance with Eqs. (4.6) and (4.7). Terms $\sum_{i=1}^{w} i$ and $\sum_{i=1}^{w} i^2$ are constants and terms $\sum_{i=1}^{w} t_i$ and $\sum_{i=1}^{w} it_i$ can be computed online, similar to the Streaming Sum and Mean hardware modules depicted in Figure 5.4.



(a) Linearity (b) Convex

Figure 5.13: Linearity and Concave hardware modules.



0
1
w
T
...
Window Length w
Running T Sum Register
Running Index * T Sum Register
Oldest Datapoint Register
+
X
-
-
T Sum Register
+
Constant Index Sum Register
Constant Index ^2 Sum Register
X
Index * T Sum Register
X
Numerator 1st b term Register
Constant Denominator Register
Numerator 2nd term Register
-
-
X
÷
÷
Numerator 1st a term Register
a Register
b Register
I
X
+
Linear Approximation
L

Figure 5.14: Linear Approximation hardware module.

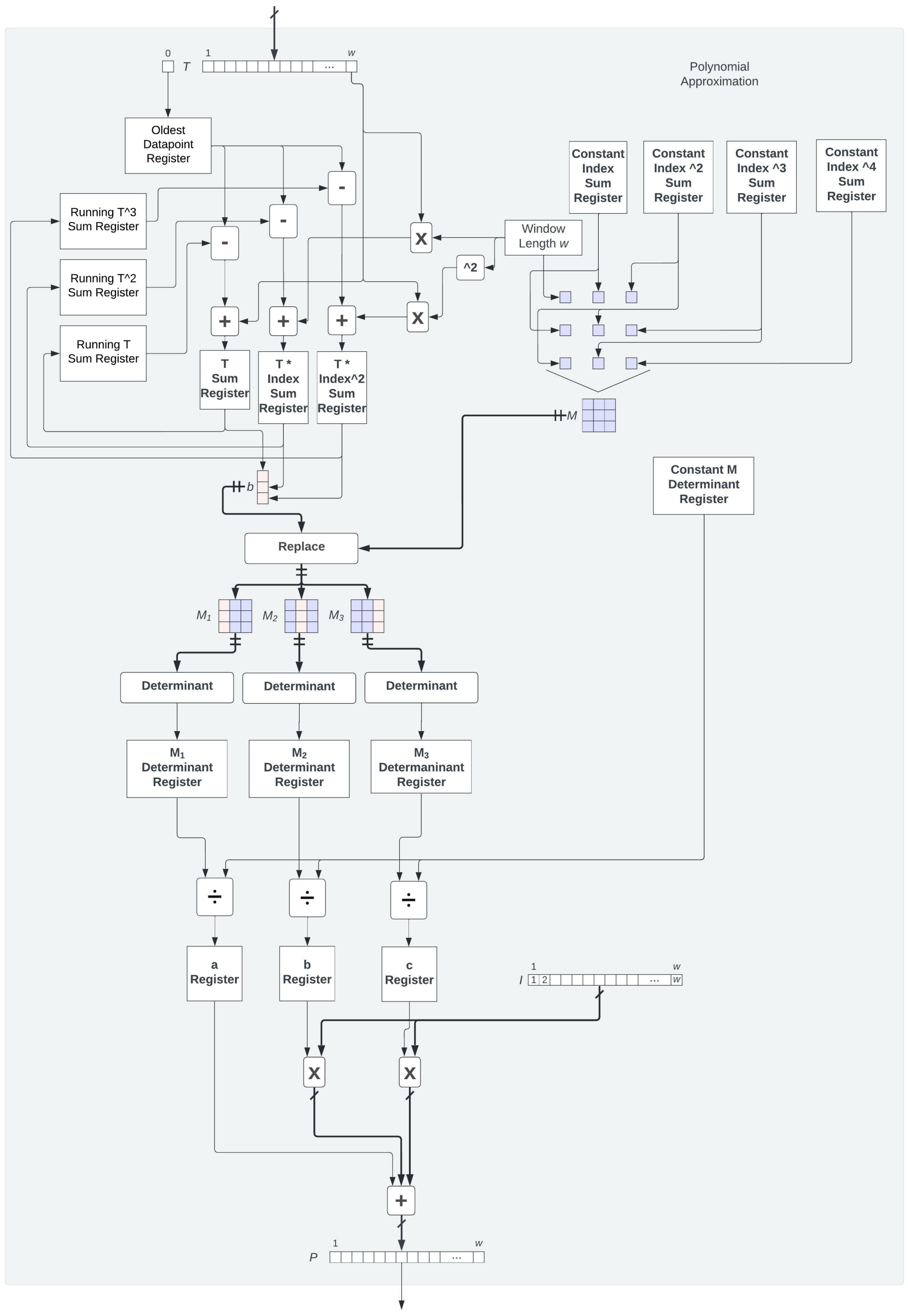
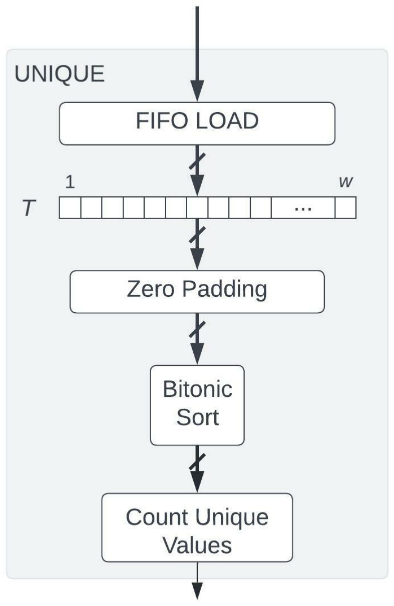


Figure 5.15: Quadratic Approximation hardware module.

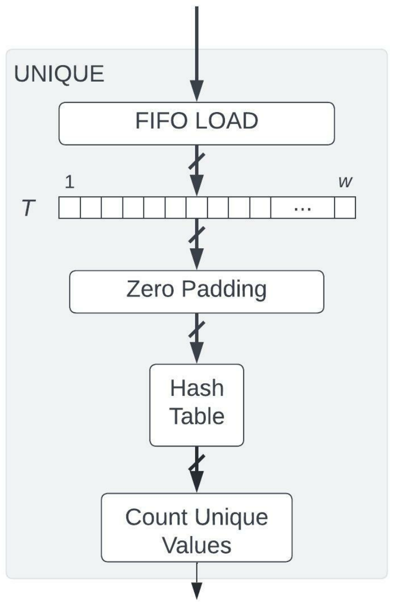
Figure 5.15 depicts the hardware module that computes the quadratic approximation of a time series via polynomial regression in accordance with Eqs. (4.8) and (4.9). In Eq. (4.8), matrix M consists of constant values that be computed offline; likewise, M's determinant, $det(M)$ can be computed offline; and vector $\mathbf{b}$ consists of sums (terms: $\sum_{i=1}^{w} t_i$, $\sum_{i=1}^{w} it_i$, and $\sum_{i=1}^{w} i^2 t_i$) that can be computed online, similar to the Streaming Sum and Mean hardware modules depicted in Figure 5.4. The bulk of the computation is spent computing determinants of three 3×3 matrices at runtime and then dividing each by constant $det(M)$ (Eq. (4.9).

5.2.11 Unique Patterns

Figure 5.16 depicts two hardware modules that compute the **Unique Pattern** feature score (Figure 5.1). The hardware module shown in Figure 5.16a is true to the NLP Time Series Search reference code, which sorts the time series in $O(nlogn)$ time and then makes a linear pass over the sorted vector to count the number of values that occurs exactly once; our implementation uses a Bitonic Sort hardware module provided by Xilinx as an IP. Figure 5.16b depicts a more efficient approach, which employs a hash table rather than a sorting algorithm to identify unique values. While the hash table approach has a time complexity of $O(n^2)$ (i.e., in the worst case, all values hash to the same bucket), it's average-case time complexity is linear, making it much more efficient in practice. We compare the performance of both implementations in the next Chapter.



(a) Unique Pattern hardware module (Bitonic Sort Implementation)



(b) Unique Pattern hardware module (Hash Table Implementation)

Figure 5.16: Unique Pattern hardware modules.

5.2.12 Periodic

Figures 5.17 and 5.18 depict two hardware modules that compute the **Periodic** feature score (5.2). Both modules compute the difference between the input time series (which has already been z-score normalized, as per Figure 5.2) and an approximation modeled by two FFT coefficients. The approximation is computed by transforming the (z-score normalized) time series into the frequency domain via FFT, extracting the four largest coefficients, placing them in-order in the first four positions in the vector, zeroing out the remaining entries, and transforming the approximation back to the time domain via IFFT.

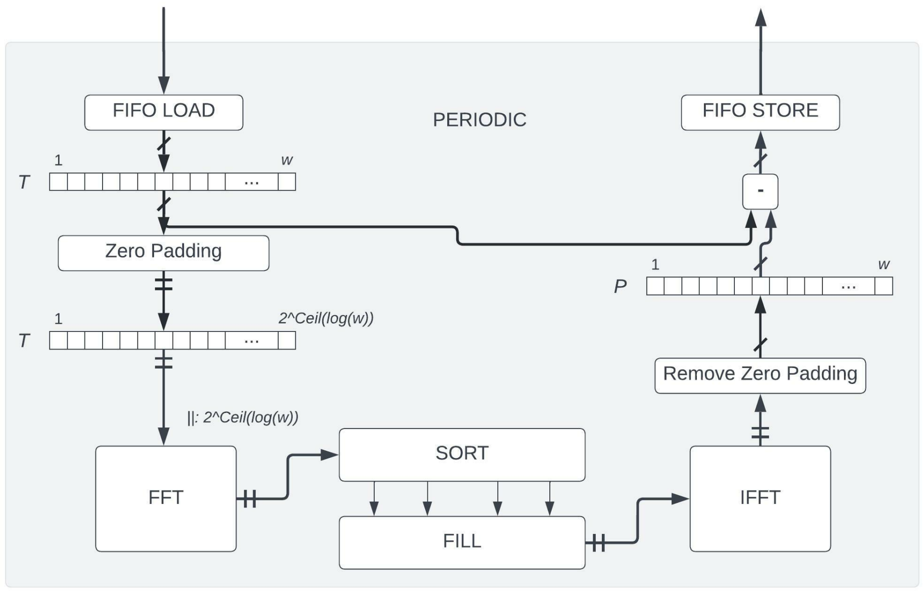


Figure 5.17: Periodic hardware module (Bitonic Sort Implementation).

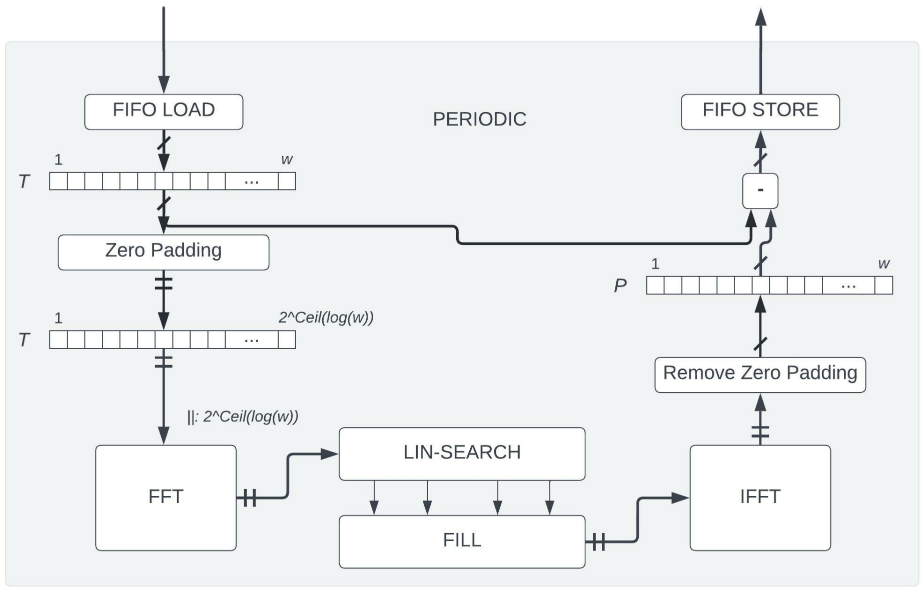


Figure 5.18: Periodic hardware module (Linear Search Implementation).

The hardware module shown in Figure 5.17 is true to the NLP Time Series Search reference code, which sorts the FFT coefficients in $O(nlogn)$ time and then zeroes out all but the top-four coefficients in-place; our implementation uses a Bitonic Sort hardware module provided by Xilinx as an IP. Figure 5.18 depicts a more efficient approach, which identifies the four maximum values in the coefficient vector using an $O(n)$ time linear search.

5.3 Normalization and Invert Modules

This section summarizes the Normalization and Invert modules shown on the bottom of Figure 5.1: the eight parallel regions compute fourteen feature scores, which are buffered in BRAM and then normalized. Each score buffer is normalized using one of four normalization algorithms as specified in Table 4.1, generating fourteen feature vectors. In accordance with Table 4.1, the Invert module computes the corresponding paired feature vector for eleven of the fourteen feature vectors previously normalized.

5.3.1 Min-Max Normalization ($\mathcal{N}_{min-max}$)

Figure 5.19 presents the hardware architecture for Min-Max Normalization (Eq. 3.4), which is applied to ten of the fourteen score buffers produced by the eight parallel regions (Table 4.1).

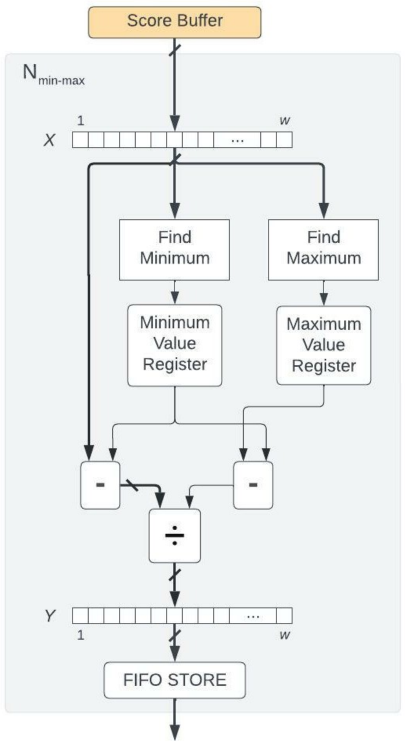


Figure 5.19: Min-Max Normalization ($\mathcal{N}_{min-max}$) hardware module

5.3.2 Min-Max Normalization with Minimum Transform ($\mathcal{N}_{min-max}^{min-trans}$)

Figure 5.20 presents the hardware architecture for Min-Max Normalization with Minimum Transform (Eq. 3.5); this normalization method is only applied to the **Low** score buffer.

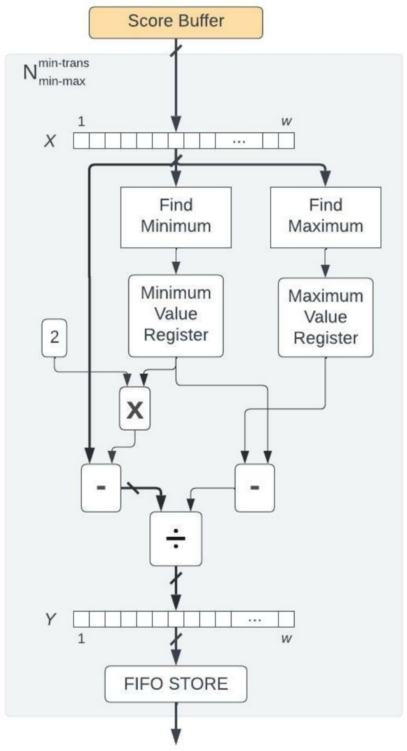


Figure 5.20: Min-Max Normalization with Minimum Transform ($\mathcal{N}_{min-max}^{min-trans}$) hardware module

5.3.3 Min-Max Normalization Scaled by the Maximum ($\mathcal{N}_{min-max}^{max-scaled}$)

Figure 5.21 presents the hardware architecture for Min-Max Normalization Scaled by the Maximum (Eq. 3.6); this normalization method is only applied to the **Rising** score buffer.

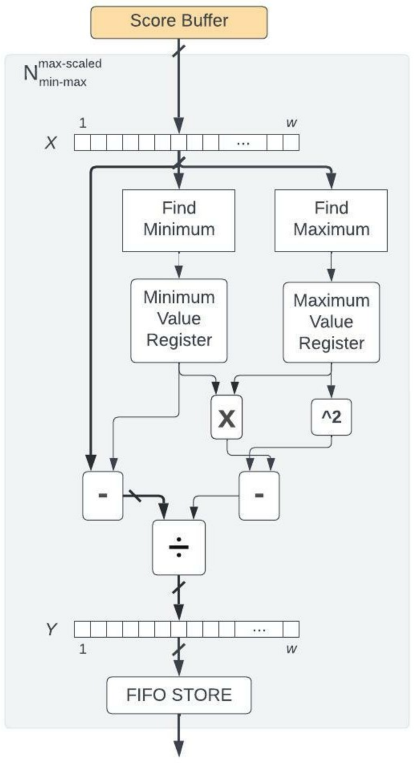


Figure 5.21: Min-Max Normalization Scaled by the Maximum ($\mathcal{N}_{min-max}^{max-scaled}$) hardware module.

5.3.4 Min-Max Normalization with Positive Guarantee ($\mathcal{N}_{min-max}^{+}$)

Figure 5.22 presents the hardware architecture for Min-Max Normalization Scaled by the Maximum (Eq. 3.7); this normalization method is only applied to the **Symmetric** and **Step** score buffers.

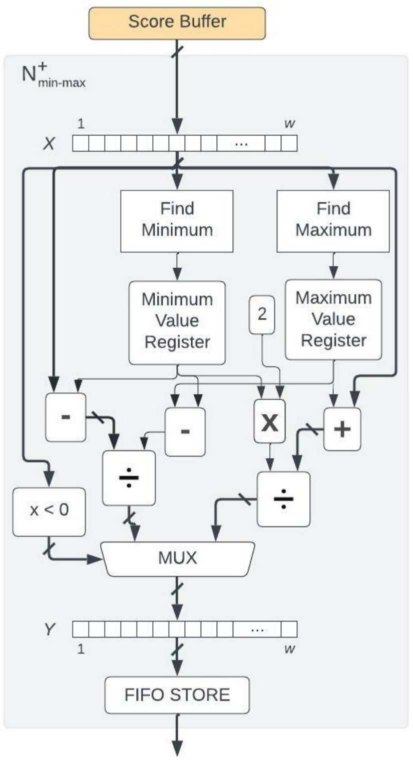


Figure 5.22: Min-Max Normalization with Positive Guarantee ($\mathcal{N}^{+}_{min-max}$) hardware module.

5.3.5 Invert

Figure 5.23 presents the hardware architecture for the Invert hardware block that generates paired feature vectors: $F_{paired\_feature} = \mathbf{1}_w - F_{feature}$. The operation is a fully unrollable constant subtraction vector operation. Eleven of the fourteen feature vectors produced by the accelerator (Figure 5.1) have paired features (Table 4.1).

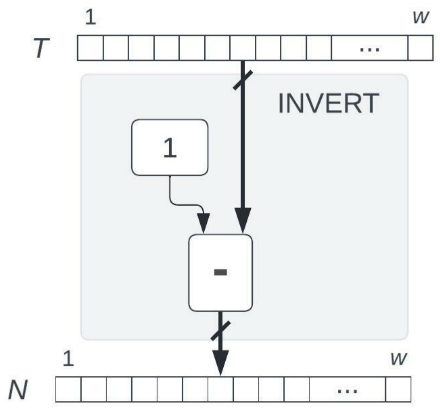


Figure 5.23: INVERT HW Module

Chapter 6

Experimental Results

6.1 Experimental Setup

The NLP Time Series Search paper [12] includes a project website[1], which provides source code written in MATLAB. A former UCR undergraduate student (Julian Beaulieu) rewrote the NLP Time Series Search program in Python and provided a reference copy for use[2]. For this project, the NLP Time Series Search's **Feature Extraction** source code was rewritten in C/C++ to be compatible with commercial high-level synthesis tools. The C/C++ source code was compiled using Vivado HLS 2018.3 and Vitis HLS 2020.2, targeting a Xilinx Zedboard (for initial testing) and a larger-scale Alveo U280 card for performance evaluation; all results reported here are for the Alveo U280 card.

[1]`https://sites.google.com/site/nlptimeseries`

[2]To the best of my knowledge, Julian has not published his source code publicly.

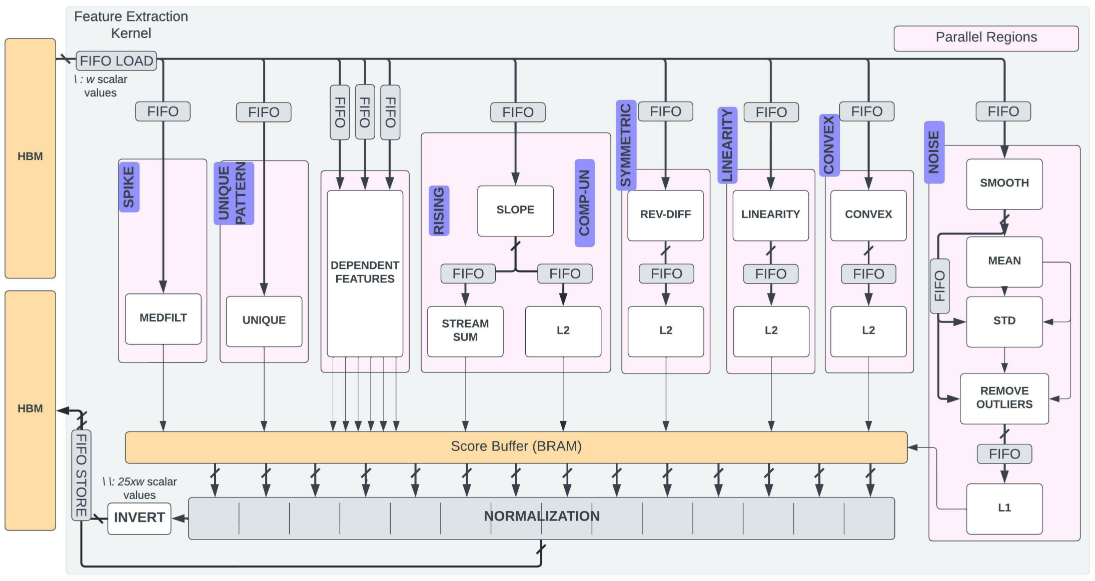


Figure 6.1: FPGA Accelerator Architecture with (HBM).

In accordance with the Vitis documentation, we use the term “kernel” to refer to the compute-intensive portion of an algorithm that is accelerated on the FPGA; in our case, the entire NLP Time Series Search implementation is itself a kernel. We set the datatype to floating-point and used a window size of 100 elements, matching the design choices used in the reference code. To mimic a streaming sensor, we wrote a host-program to preload a time series into the Alveo U280 card’s High Bandwidth Memory (HBM), which has an 8 GB capacity, and to stream the data from the HBM to the FPGA. Figure 6.1 demonstrates this setup, which replaces the sensor and host shown in Figure 5.1 with reads from and writes to HBM.

We used the `HLS DATAFLOW` pragma to enable the eight parallel regions in Figure 5.1 to execute concurrently, and within each feature extractor to enable further concurrency; additionally, we used the `HLS PIPELINE` pragma within each feature extractor to allow concurrent execution of operations within a loop across iterations; each module is organized in a load-compute-store format in order to overlap computation with memory accesses and data transfers.

The interconnects that transfer subsequences between the modules are instantiated as HLS Streams, which transmit data values sequentially; the alternative would be to transmit data values via buffers, which would require address generation for each read and write. Each hardware module reads data elements from the stream and stores its values in a local array for computation; when computation finishes, the module writes its output array into an outgoing stream in sequential order. The streams are implemented as FIFOs in hardware. Some modules output single scalar values rather than arrays/streams, which are implemented as regular data ports without protocol support. The depth of each HLS stream is set to w, the window size.

Each parallel region in the design operates at its own clock frequency. The latency to process a window of datapoints is dominated by the longest latency among the feature extractors. Four of the parallel regions that compute feature scores (**Periodic**, **Unique Pattern**, **Step** and **Noise**) have data-dependent latencies; the underlying data-dependent hardware modules are the Bitonic Sort IP (used by **Periodic** and **Unique Pattern** (see Figures 5.17 and 5.16a) and the Remove_Outliers hardware module used by **Step** and **Noise**). **Periodic** and **Unique Patterns** become fixed-latency when the Bitonic Sort IP is replaced with a Linear Search (Figure 5.17) and

Hash Table (Figure 5.16b) respectively; coincidentally, the **Periodic** and **Unique Patterns** feature score computes have the two highest overall latencies as well.

Prior to execution, a window consisting of all zero values is transmitted through the entire accelerator to initialize all streaming sums; all feature scores at this point are invalid and are discarded prior to direct execution on the first window of sampled datapoints.

6.2 Baseline Results

Our baseline design implements the **Periodic** and **Unique Patterns** feature extractors using the Bitonic Sort IP module. In this design, **Periodic** has the highest overall latency among all feature extractors, and therefore determines the kernel's latency. Table 6.1 summarizes the accelerator kernel's performance, including frequency, maximum and minimum latencies, and usage of different types of FPGA resources: flip-flops (FF), Lookup-Tables (LUT), DSP Blocks (DSP), and Block RAMs (BRAM). The accelerator kernel used 10.42% of the FPGA's register capacity, while using 20.28% of available LUTs, 12.45% of available BRAMs and 3.48% of available DSP blocks. The 346 MHz frequency reported in Table 6.1 is for the kernel's top-level module (”Feature Extraction Kernel” in Figures 5.1 and 6.1); each feature extraction kernel ran at its own frequency.

Each feature extractor is divided into two parallel steps: feature score computation and normalization, with performance splits shown in Table 6.2, which reports the

Table 6.1: Design characteristics of the prototype FPGA kernel

Feature Extraction Kernel						
Frequency	Min. Latency (cycles)	Max. Latency (cycles)	FF	LUT	DSP	BRAM
346 MHz	16,693	30,676	271,420	264,201	314	251

Table 6.2: Latency splits of the full system accelerator

Performance Splits of Kernel				
	Min. Latency (cycles)	Max. Latency (cycles)	Min. Latency (μs)	Max. Latency (μs)
Computation	16,511	26,836	55.0	89.4
Normalization	1,079	1,079	3.60	3.60
Full Kernel	16,693	30,676	55.6	102

latency required to extract features from and normalize a full window of $w = 100$ data points. Referring back to Figures 5.1 and 6.1, 8 parallel regions generate 14 features, which are then normalized. Recall that each feature extractor generates w feature scores, which are then normalized; thus, normalization executes $1/w$ as frequently as the modules that generate the feature scores. The percentage of time allocated to the computation of feature scores ranges from 87% (maximum latency) to 99% (minimum latency). In addition to the normalization step, some additional latency (not reported in Table 6.2) is spent updating static registers, writing buffer scores to BRAM, and writing back the results to global memory.

6.3 Feature Extractor Performance

This section will summarize the performance of the different feature score hardware modules. We begin by discussing performance optimizations applied to **Periodic** and **Unique Patterns**, and then we compare their performance to the other feature score modules. Prior to optimization, **Periodic** had the highest latency and constraints the latency of the entire design. After optimizing **Periodic**, **Unique Patterns** had the highest latency. After optimizing **Unique Patterns**, the now-optimized **Periodic** had the highest overall latency. We could not determine any subsequent optimizations that could further reduce the latency of **Periodic**, and optimizing the latency of other hardware modules could not yield any additional system-level performance improvements.

We applied the `HLS UNROLL` pragma to both **Periodic** and **Unique Patterns**. **Unique Patterns** features a single loop that can be fully unrolled. All loops within **Periodic** were fully unrolled as well, except for the loops that read and write to the FIFOs, where unrolling will not benefit performance.

6.3.1 Periodic

The latency of the baseline implementation of the **Periodic** feature score hardware module was dominated by the latency of the FFT (6332 cycles) and a data-dependent bitonic sorter (3096 cycles in the best case; 13,421 cycles in the worst-case). Taken together, the two optimizations described below reduced **Periodic**'s latency from

54.811 μs (best case) and 89.224 μs (worst case) to 9.993 μs constant (not data-dependent) latency.

FFT: We implemented the FFT using a Xilinx IP module, which employs a default streaming pipeline architecture that is optimized for performance at the cost of a considerable amount of area. We do not have the ability to optimize or modify the FFT IP directly to further improve performance, for example using pragmas. The FFT employs two wrappers, FFT FWD and FFT INV, which set up data prior to calling the FFT IP to perform either a forward FFT or an inverse IFFT. To optimize the wrappers, we fully unrolled the data transfer loop; to ensure that the length of the input to the FFT IP has a power of 2, we added a second concurrent loop to zero-pad the input and unrolled it fully as well. A second optimization is to switch from a Radix-2 FFT to a Radix-4 FFT.

Table 6.3 compares the performance of the different FFT implementations. As a baseline for comparison, we took the open-source Project Nayuki FFT[3] and synthesized it using HLS. While the Project Nayuki FFT performed incrementally better than the Radix-2 Xilinx FFT IP with an optimized wrapper, the Radix-4 Xilinx FFT IP ran an order of magnitude faster. Based on this comparison, we selected the Radix-4 FFT IP with optimized wrapper to accelerate the Period feature extractor.

Bitonic Sort: As mentioned earlier, the NLP Time Series Search reference code employs an $O(nlogn)$ sorting algorithm within the **Periodic** feature extractor; however,

[3]`https://www.nayuki.io/page/free-small-fft-in-multiple-languages`

Table 6.3: Performance of Different FFTs.

FFT Comparison		
FFT Implementation	Latency (cycles)	Latency (μs)
Xilinx Radix-2 FFT + Unoptimized Wrapper	6,385	21.3
Xilinx Radix-2 FFT + Optimized Wrapper	6,332	21.1
Project Nayuki FFT Compiled via HLS	5,702	19.1
Xilinx Radix-4 FFT + Unoptimized Wrapper	972	3.24
Xilinx Radix-4 FFT + Optimized Wrapper	849	2.83

Table 6.4: Performance comparison between the Xilinx Bitonic Sort IP and Linear Search.

Bitonic Sort vs. Linear Search				
	Min. Latency (cycles)	Max. Latency (cycles)	Min. Latency (μs)	Max. Latency (μs)
Bitonic Sort	3,096	13,421	10.3	44.7
Linear Search	853	853	2.84	2.84

only the four maximum values in the sorted list are retained, and the other values are discarded. Our initial implementation of the Periodic feature extractor tried to be as faithful as possible to the reference code, with the main difference being the choice of sorting algorithm. We used a Bitonic Sort IP provided by Xilinx, which consumed around 50% of the full kernel execution time under the worst-case latency assumptions. A more efficient approach was to employ an $O(n)$-time linear search that identifies the four largest-valued datapoints within the window, noting that doing so could also optimize the software performance. As shown in Table 6.4, this change reduced the latency to 853 cycles, and eliminated data-dependent latency,

Table 6.5: Performance comparison between the Xilinx Bitonic Sort IP and a Hash Table.

Bitonic Sort vs. Hash Table Search				
	Min. Latency (cycles)	Max. Latency (cycles)	Min. Latency (μs)	Max. Latency (μs)
Bitonic Sort	3,096	13,421	10.3	44.7
Hash Table	501	501	1.67	1.67

6.3.2 Unique Patterns

As mentioned earlier, NLP Time Series Search reference code employs an $O(nlogn)$ sorting algorithm within the **Unique Patterns** feature extractor; here, the purpose of the sorting algorithm is to count the number of datapoints in the time series with unique values, which is trivial when the time series is sorted. That said, a more efficient approach is to employ a hash table, rather than a sorting algorithm, which could work as a software optimization as well. As shown in Table 6.5 the Bitonic Sort IP with a hash table reduced the latency of the Unique Feature Extractor from 10.746 μs (best case) and 45.159 μs (worst case) by an order of magnitude to 1.670 μs constant (not data-dependent) latency.

6.3.3 Feature Score Hardware Module Comparison

Figures 6.2 and 6.3 report the latencies of each of the feature score computation hardware modules in terms of cycle count and time (μs). The highest-latency module is **Periodic**, which constrains the overall latency of the design; the other hardware modules need to wait for **Periodic** to finish once they have computed their feature scores.

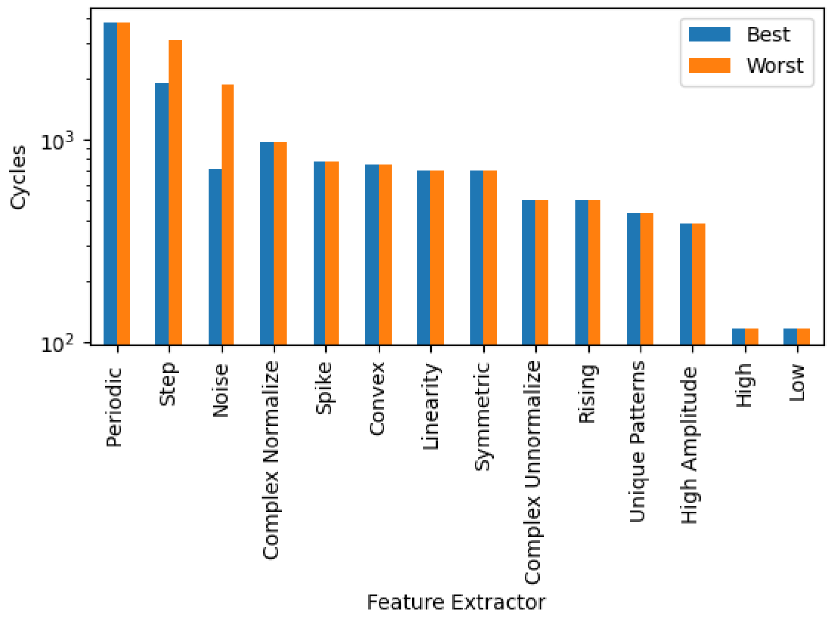


Figure 6.2: Latency (cycles) of each feature score hardware module.

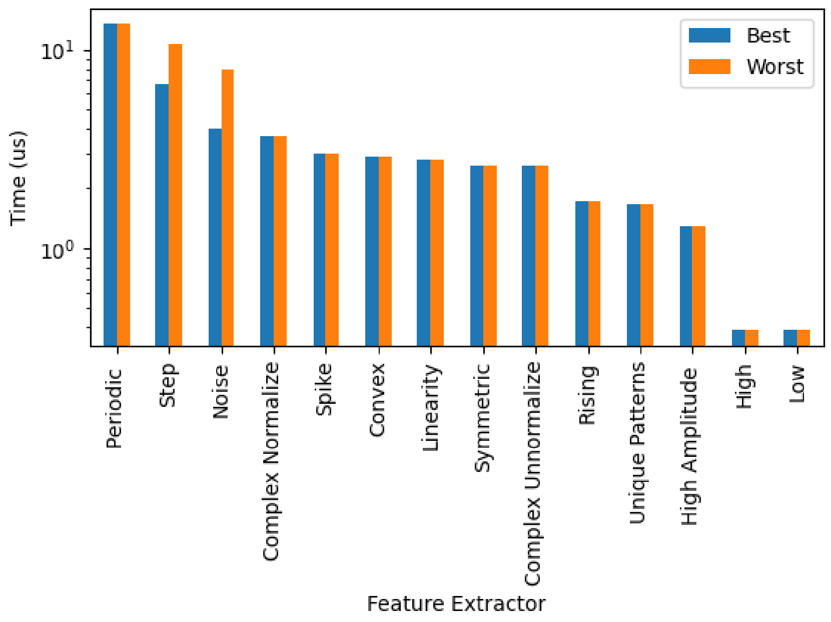


Figure 6.3: Latency (time) of each feature score hardware module.

As discussed earlier, the **Periodic** and **Unique Patterns** hardware modules have fixed-latency, due to removal of the Bitonic Sort IP. The **Step** and **Noise** hardware modules have variable latencies, due to the Remove_Outliers hardware module.

Each arithmetic hardware module operates at its own frequency. Figure 6.4 reports the respective frequencies of each of the feature score extractors. Frequencies range from $\tilde{3}50$ MHz (**Periodic**, **Step**, **Complex Normalize**) to $\tilde{4}25$ MHz. It may be possible to achieve higher performance by increasing the frequency of the **Periodic** and **Step** hardware modules, as they have the two highest latencies in terms of time, as reported in Figure 6.3.

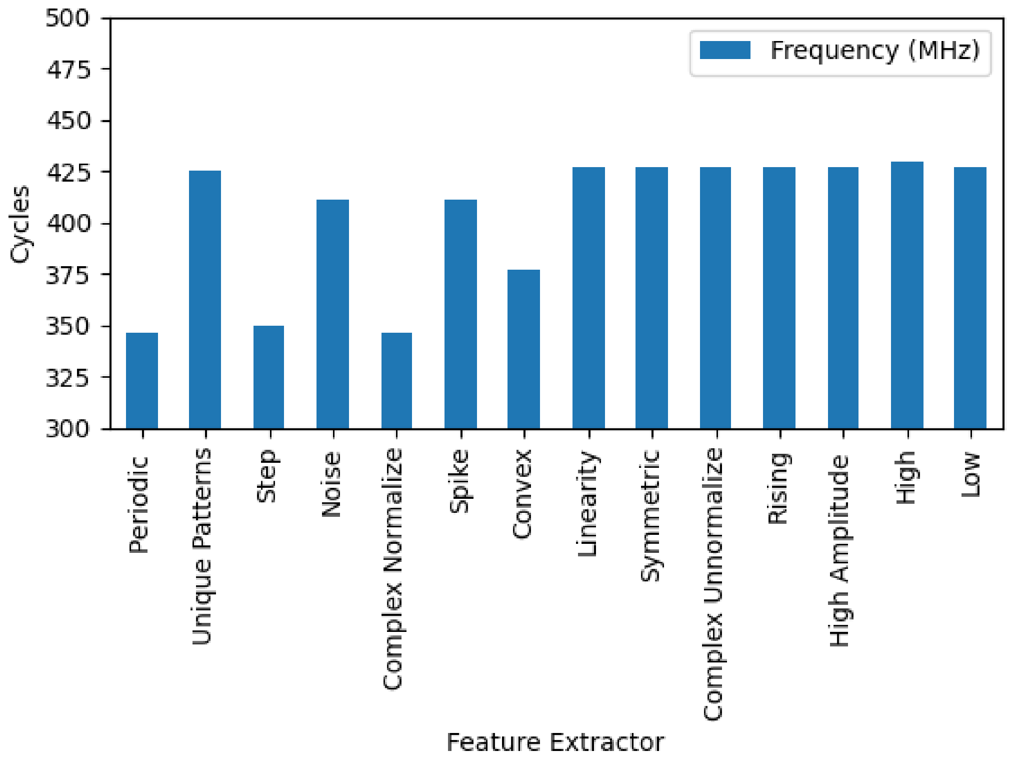


Figure 6.4: Frequency of each feature score hardware module.

6.4 Normalization

Table 6.6 report the latencies of the different components within the normalization kernel. The latency of loading feature score vectors from BRAM exceeds the latencies of the normalization computations. The latency of the Invert module, which is used to generate feature pairs, is not particularly significant. The critical path runs through the $\mathcal{N}^{+}_{min-max}$ normalization method and includes the Invert module.

Table 6.6: Latencies of hardware modules within the Normalization kernel.

Performance Splits of Normalization		
	Latency (cycles)	Latency (μs)
Load from BRAM	707	2.36
$\mathcal{N}_{min-max}$	443	1.48
$\mathcal{N}_{min-max}^{min-trans}$	448	1.49
$\mathcal{N}_{min-max}^{max-scaled}$	333	1.11
$\mathcal{N}_{min-max}^{+}$	478	1.59
Invert	102	0.340

6.5 FPGA vs GPU

This section compares the performance and power consumption of two-based FPGA implementations of NLP Time Series Search to a GPU-based implementation, which we also wrote. Our second FPGA-based implementation is based on the same hardware architecture as described previously, but was compiled with Vitis under a power optimization, setting.

The GPU kernel that was created for comparing the FPGA design was built with CUDA C and was executed on an Nvidia GeForce RTX 2080 Ti GPU. The GPU implements the offline NLP Time Series Search Algorithm, not the online version, evaluated using the FPGA. The GPU computes feature scores sequentially, one-by-one; each feature score computational algorithm is parallelized for maximal throughput. Feature score vectors are written back to memory. The GPU then normalizes each feature score vector in an offline fashion; i.e., each feature score vector is fully normalized, rather than decomposing the feature score vector into length-w subvectors, each of which is then normalized.

Table 6.7: FPGA Kernel vs GPU Kernel

FPGA vs GPU			
	Latency (ms)	Throughput (FLOPS)	Power (W)
FPGA (Performance Optimized)	220	5.8×10^6	32.1
GPU	244	5.0×10^6	52.0
FPGA (Power Optimized)	260	4.8×10^6	30.7

Table 6.7 compares the performance (latency and throughput) and power consumption of the Performance- and Power-optimized FPGA implementations to that of the GPU. The performance-optimized FPGA implementation achieves the lowest latency and the highest throughput overall, including outperforming the GPU by 24 ms (latency) and 0.8 FLOPS (throughput), and reducing power consumption by 19.9 W. An additional 1.4 W of power can be saved by switching to the power-optimized FPGA implementation, although doing so yields a design that is inferior to the GPU in terms of performance metrics (higher latency, lower throughput).

Chapter 7

Conclusions

In this paper a feature extraction FPGA kernel was created using C++ and Xilinx's Vitis HLS. The design is able to extraction 25 features in real-time by streaming data directly into the computation architecture. The design was performance tested and optimized by reducing the latency of bottleneck hardware modules. The current design can be further developed to prioritize different attributes. If the design needs to be optimized for latency, we would keep attacking the bottlenecks of the design and use more of the available hardware resources to improve performance. Another avenue of development would be to optimize for hardware utilization. By simplifying all the hardware modules' architecture slowing them down to the same speed as the bottleneck, we would achieve the same latency but at a lower resource utilization cost. Lastly, we could try to lower frequency of the hardware modules to try to lower power consumption.

The prototype design was compared against a GPU implementation, further comparisons can be made in the future to multi-threaded CPU applications. A through study could be conducted by creating the optimal FPGA, GPU and CPU design for the kernels and comparing them in a systematic manner. Other research related to this design that can be done is to compute features in real-time and use them to perform data analysis techniques such as searches, and classifications in real-time. We could try to extract even more features, or choose from a different set of features like the Catch-22 set [15]. To extrapolate from that idea, we could create an autotunning software that takes in a user's requested feature set and outputs an FPGA design optimized to extract those specific features.

Bibliography

[1] Gustavo E. A. P. A. Batista, Eamonn J. Keogh, Oben M. Tataw, and Vinicius M. A. Souza. Cid: an efficient complexity-invariant distance for time series. *Data Mining and Knowledge Discovery*, 28:634–669, 2013.

[2] Zehua Chen, Yirui Wu, Jiang Mei, Jiamin Lu, Yunfeng Wang, and Jun Feng. Spatial-temporal motif discovery with variable-size sliding windows. In *2021 International Conference on Wireless Communications and Smart Grid (ICWCSG)*, pages 126–133, 2021.

[3] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 288–298, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Jianyi Cheng, John Wickerson, and George A. Constantinides. Probabilistic optimization for high-level synthesis. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 145, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. Rapid cycle-accurate simulator for high-level synthesis. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 178–183, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, page 493–498, New York, NY, USA, 2003. Association for Computing Machinery.

[7] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing

Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, and Gustavo Batista. The ucr time series classification archive, October 2018.

[8] Ben D. Fulcher and Nick S. Jones. Highly comparative feature-based time-series classification. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3026–3037, 2014.

[9] Konstantinos Georgopoulos, Grigorios Chrysos, Pavlos Malakonakis, Antonis Nikitakis, Nikos Tampouratzis, Apostolos Dollas, Dionisios Pnevmatikatos, and Yannis Papaefstathiou. An evaluation of vivado hls for efficient system design. In *2016 International Symposium ELMAR*, pages 195–199, 2016.

[10] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. Rapidstream: Parallel physical implementation of fpga hls designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 1–12, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-mei Hwu. Pylog: An algorithm-centric python-based fpga programming and synthesis flow. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 227–228, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Shima Imani, Sara Alaee, and Eamonn Keogh. Putting the human in the time series analytics loop. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 635–644, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Amin Kalantar, Zachary Zimmerman, and Philip Brisk. Fa-lamp: Fpga-accelerated learned approximate matrix profile for time series similarity prediction. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–49, 2021.

[14] Reza Lotfian and Roozbeh Jafari. An ultra-low power hardware accelerator architecture for wearable computers using dynamic time warping. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 913–916, 2013.

[15] Carl Henning Lubba, Sarab Sethi, Philip Knaute, Simon Schultz, Ben Fulcher, and Nick Jones. catch22: Canonical time-series characteristics: Selected through highly comparative time-series analysis. *Data Mining and Knowledge Discovery*, 33, 08 2019.

[16] Leonard MacEachern and Ghazaleh Vazhbakht. Configurable fpga-based outlier detection for time series data. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 142–145, 2020.

[17] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[18] Jose Carlos Romero, Angeles Navarro, Antonio Vilches, Andrés Rodríguez, Francisco Corbera, and Rafael Asenjo. Efficient heterogeneous matrix profile on a cpu + high performance fpga with integrated hbm. *Future Gener. Comput. Syst.*, 125(C):10–23, dec 2021.

[19] Doruk Sart, Abdullah Mueen, Walid Najjar, Eamonn Keogh, and Vit Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *2010 IEEE International Conference on Data Mining*, pages 1001–1006, 2010.

[20] Xiaozhe Wang, Anthony Wirth, and Liang Wang. Structure-based statistical features and multivariate time series clustering. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 351–360, 2007.

[21] Zilong Wang, Sitao Huang, Lanjun Wang, Hao Li, Yu Wang, and Huazhong Yang. Accelerating subsequence similarity search based on dynamic time warping distance with fpga. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, page 53–62, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 78–88, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Jidong Yuan, Qianhong Lin, Wei Zhang, and Zhihai Wang. Locally slope-based dynamic time warping for time series classification. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 1713–1722, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth Funning, Philip Brisk, and Eamonn Keogh. Matrix profile xiv: Scaling

time series motif discovery with gpus to break a quintillion pairwise comparisons a day and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 74–86, New York, NY, USA, 2019. Association for Computing Machinery.