Imperial College
London

Imperial College London

Department of Mathematics

Solving the Collatz conjecture

Author: Ilia Sobakinskikh (CID: 00000000)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2022-2023

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

This is where you usually thank people who have provided useful assistance, feedback,...., during your project.

Abstract

The abstract is a short summary of the thesis' contents. It should be about half a page long and be accessible by someone not familiar with the project. The goal of the abstract is also to tease the reader and make him want to read the whole thesis.

Contents

List of Figures

List of Tables

Introduction

In this thesis, we explore how the inference time of a Transformer Neural Network can be efficiently optimized with applications to real-time anomaly detection in financial time series. The financial time series are price series such as asset prices. Unfortunately, the data is often with errors or outliers that make the downstream data processing tasks useless, unstable or even harmful [4] [5]. Moreover, the amount of financial time-series data has been significantly increasing [6]. Hence, there is a need for better data-cleaning methods in terms of accuracy and in terms of processing speed.

Transformers as a neural network architecture have achieved superior performances in many tasks such as Natural Language Processing and Computer Vision [7]. Time series modelling and especially anomaly detection tasks can benefit from the features of transformers architecture in multiple ways, including the capacity to capture long-range dependencies and interactions [8].

Increasingly powerful hardware, such as field-programmable gate arrays (FPGAs), have seen increasing usage in recent years due to their reconfigurability and high performance [9].

We explore different Transformer architectures for time series modelling and how they can be efficiently implemented on an FPGA board (PYNQ-Z2). In particular, we examine the application of Transformers to detect anomalies in time series and we show how they can be efficiently implemented on an FPGA board to minimize latency or to maximize throughput.

Chapter 1

Methodology

In this chapter, we will describe the main concepts and ideas used in this work. The reader will be introduced to the main concepts of the Transformer architecture and how it can be used for anomaly detection in time series. Finally, the main concepts of programming an FPGA will be introduced and the specific optimizations that can be applied to speed up the computations.

1.1 Problem Formulation

Definition 1.1.1. We consider a **time-series** $\mathcal{T}$ which is simply a timestamped sequence of observations $x_i \in R^n$.

Remark 1.1.2. Most of the times we will consider univariate case, i.e. $n = 1$. An example of this is a price time-series of a single stock. However, the multivariate case is also important and we will consider it in the experiments. For example, one can consider a time-series of prices of multiple stocks to get a multivariate time series.

Definition 1.1.3. The **Anomaly Detection** task: for any time-series $\hat{\mathcal{T}}$ of length n, we need to predict $\mathcal{Y} = \{y_1, ..., y_n\}, y_i \in \{0, 1\}$, whether the datapoint at the i-th timestamp anomalous (where by convention we will use 1 as anomaly and 0 as not an anomaly).

In this work, we will restrict ourselves to the **supervised case** where the labels y_i are known for the *seen* (or training) part of the dataset.

Remark 1.1.4. One can also consider an unsupervised task. However, one issue with the unsupervised task is that it is hard to evaluate the performance (i.e., accuracy) of the model's predictions [10].

1.2 Transformers

In this section, we will describe the main concepts of the Transformer architecture. We will describe the main building blocks of the Transformer architecture and will give a special treatment to the attention mechanism firstly introduced in [11].

General architecture

In [1], authors introduced the Transformer architecture which a neural network architecture which is the architecture that is dominantly used in Natural Language Processing tasks. The architecture's main feature was reliance on the attention mechanism and the complete elimination of recurrent and convolutional layers.

Figure 1.1 presents the main architecture of the transformer. The architecture consists of an **encoder** and a **decoder**. For the purpose of the thesis we will only consider the **encoder** part of the architecture. The **encoder** is preceded by a **positional encoding** layer which is used to *inject* the positional information to the input vectors x_i because the attention mechanism is permutation invariant, this will be explained in Section 1.2.1.

The **encoder** consists of N identical layers. Each layer has two sub-layers which are a **multi-head self-attention** layer and a **feed-forward** layer. The **feed-forward** layer $\mathrm{FFN}(\cdot)$ is a simple

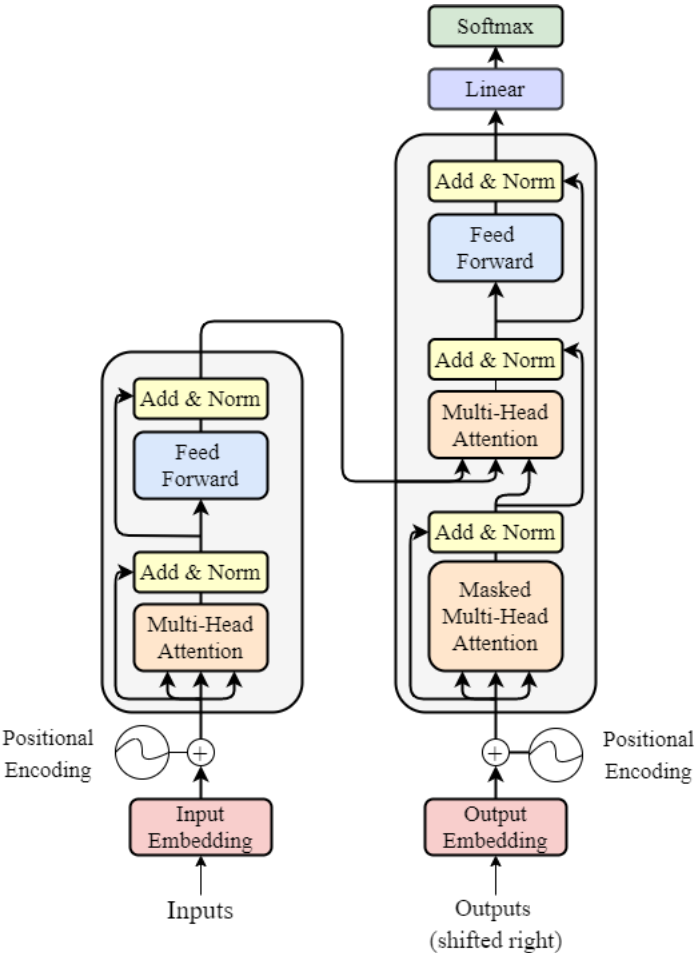


Figure 1.1: Model architecture of the Transformer [1]

fully-connected layer with a non-linear activation function applied element-wise to the result[1]. Specifically, authors of [1] used an FC layer with the ReLU activation function

$$\text{ReLU}(x) = \max(0, x)$$

followed by another FC layer without activation function, i.e.

$$\text{FFN}(x) = W_2 \cdot ReLU(W_1 \cdot x + b_1) + b_2$$

The **Add & Norm** layer is a residual connection [12] followed by a layer normalization layers [13]. Those layers are not essential for this work and will not be described in detail.

This constitutes the main building block of the Transformer encoder architecture. The next section will describe the attention mechanism in detail.

1.2.1 Attention mechanism

This section will describe the attention mechanism, its variations and the intuition behind it. Moreover, we will compare different attention mechanism implementations in terms of their computational complexity and their ability to capture long-range dependencies.

Dot-Product Attention and Multi-Head Attention

The attention mechanism introduced in the Transformer architecture [1] used a **scaled dot-product attention**.

[1] In general, a **fully-connected** layer $\text{FC}(x)$ is simply a linear transformation inputs X (i.e., a matrix multiplication) with the activation function applied element-wise to the result. That is, $\text{FC}(x) = f(W \cdot x + b)$ where W is a weight matrix and b is a bias vector and $f(\cdot)$ is the activation function. Authors of [1] used the ReLU activation function

The main idea of the **dot-product attention** mechanism is to compute the mapping of a query q_i for each input vector x_i to a set of key-value pairs (k_j, v_j). The query q_i, key k_i and value v_i vectors are simply linear transformations of the input vectors x_i, i.e., $q_i = W_Q \cdot x_i$, $k_i = W_K \cdot x_i$, $v_i = W_V \cdot x_i$ where W_Q, W_K and W_V are the weight matrices. The attention mechanism is a weighted sum of the values v_j where the weights are computed as a function of the query q_i and the key k_j. That is, $Attention(x_i) = \sum_j \alpha_{ij} v_j$ where $\alpha_{ij} = \text{softmax}(q_i \cdot v_i)$ is the weight of the j-th value v_j. In practice, the attention mechanism is computed for all the queries q_i at the same time by utilizing the following expression in matrix form:

$$\begin{aligned} Q &= W_Q \cdot X, \\ K &= W_K \cdot X, \\ V &= W_V \cdot X \end{aligned} \tag{1.2.1}$$

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V \tag{1.2.2}$$

Remark 1.2.1. In [1], authors additionally scaled the weights α_i by the square root of the dimension of the key vectors d_k. This is, however, not strictly necessary and is done for numerical stability reasons.

The **multi-head attention** mechanism is simply a concatenation of multiple attention mechanisms. That is, we can compute h different attention mechanisms in parallel and then concatenate the results. The main idea behind this is that different attention mechanisms can learn different features of the input vectors.

Figure 1.1 visualizes the attention mechanism introduced in [1].

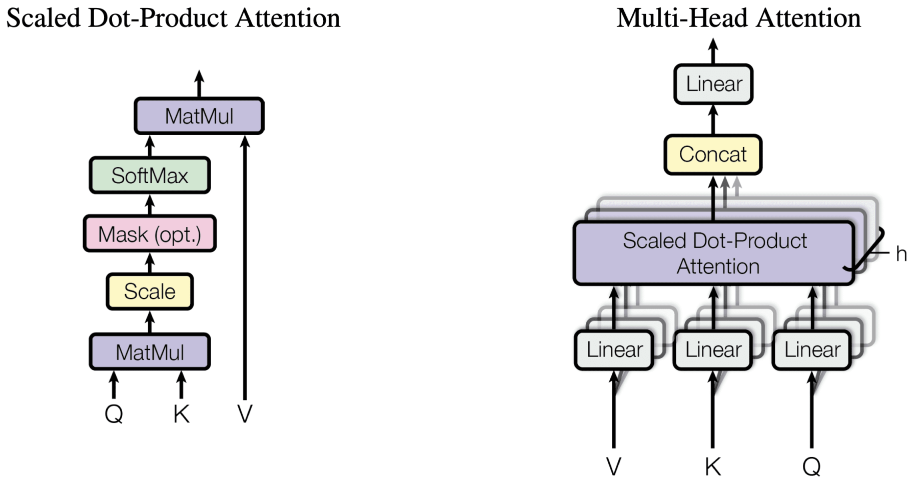


Figure 1.2: Scaled Dot-Product Attention and Multi-Head Attention [1]

Linear attention

In [14], authors propose an extension to the dot-product attention mechanism called **linear attention**. This significantly reduces the computational complexity of the attention mechanism by eliminating the need to compute the softmax function.

Notice that in 1.2.2, the softmax function is applied rowwise to the matrix QK^T. The softmax function can be substituted with a general similarity function $\text{sim}(\cdot, \cdot)$ between a query q_i and a key k_j. The equation 1.2.2 for output value v'_i can then be rewritten as follows:

$$v'_i = \text{Attention}(Q, K, V)_i = \frac{\sum_j \text{sim}(q_i, k_j) v_j}{\sum_j \text{sim}(q_i, k_j)} \tag{1.2.3}$$

The only constrained imposed on the similarity function $\text{sim}(\cdot, \cdot)$ is that it should be non-negative for it to define an attention function. This conveniently includes all kernels. That is $\text{sim}(q_i, k_j) = \phi(q_i)^T \phi(k_j)$ where $\phi(\cdot)$ is a feature map.

So that given a kernel with a feature map $\phi(\cdot)$, the attention mechanism can be computed as follows:

$$v_i' = \text{Attention}(Q, K, V)_i = \frac{\sum_j \phi(q_i)^T \phi(k_j) v_j}{\sum_j \phi(q_i)^T \phi(k_j)} \tag{1.2.4}$$

And we can rewrite the attention mechanism in matrix form as follows:

$$\text{Attention}(Q, K, V) = \frac{\phi(Q)^T \phi(K) V}{\phi(Q)^T \phi(K)} \tag{1.2.5}$$

Regrouping the terms, we get the following expression for the attention mechanism:

$$\text{Attention}(Q, K, V) = \phi(Q)^T \frac{\phi(K) V}{\phi(Q)^T \phi(K)} \tag{1.2.6}$$

which makes it evident that we can compute $\sum_j \phi(k_j) v_j$ once and reuse them for all the queries q_i which reduces the computational complexity from $O(N^2)$ to $O(N)$ where N is the number of input vectors in the attention layer.

Remark 1.2.2. In [14], authors used the $\phi(x) = elu(x)+1$ feature map where $elu(x) = \max(0, x) + \min(0, \alpha(\exp(x) - 1))$ is the exponential linear unit activation function. This feature map is used to ensure that the attention mechanism is non-negative and hence defines a valid attention function. Moreover, $elu(\cdot)$ is used instead of $ReLU(\cdot)$ to ensure the differentiability when x is negative.

1.2.2 Transformers for time series modelling

TODO: provide the main overview of the papers that use transformers for time series modelling, the comparison to other methods, the comparison of different transformer architectures specifically tailored for time-series

1.3 FPGA design

In this section, the main design principles of programming an FPGA board will be described. Readers will be introduced to the common optimization techniques and how they are achieved. The FPGA programming will be done using C++ HLS which is converted to verilog code.

1.3.1 Introduction to FPGA

The progress of hardware acceleration devices like field-programmable gate arrays (FPGAs) enables the achievement of high component density and low power consumption, all the while minimizing latency [9]. They are commonly used to accelerate high-performance, computationally intensive systems (for example, data centers) or to minimize the latency of execution (for example, in high-frequency trading).

1.3.2 FPGA development

Common Terms

In this section, common terms will be introduced. The terms will be used throughout Section 1.3.2. It is not required to read all of them at once, but it is recommended to refer to this section when a term is not clear and the reader can refer to this section when necessary.

Definition 1.3.1. LUT (Look-Up Table) is a small, fast memory that stores the output of a Boolean function of its inputs. The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables.

Definition 1.3.2. BRAM (Block RAM) is a dedicated two-port memory that can be used to store data.

Definition 1.3.3. DSP (Digital Signal Processing) is a specialized hardware unit that is optimized for performing mathematical operations on streaming data.

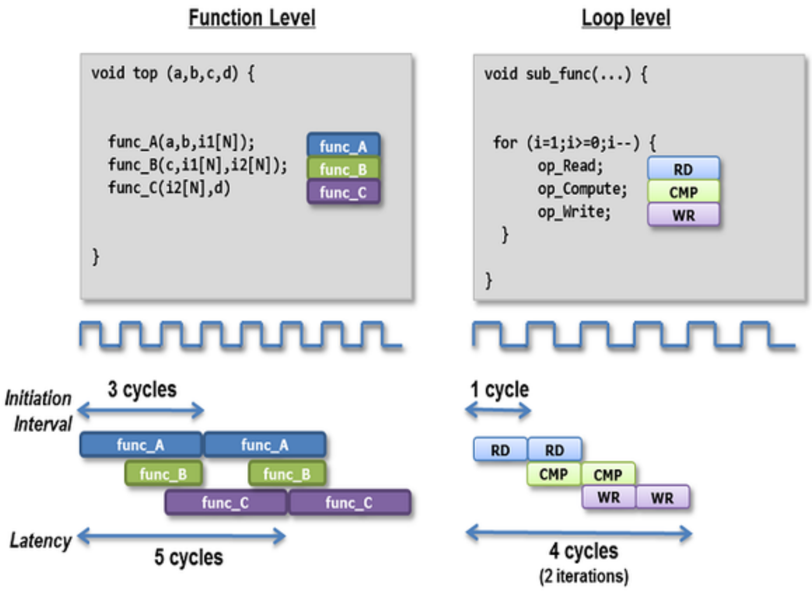
Definition 1.3.4. Clock cycle is the time between two consecutive rising edges of the clock signal. It is the amount of time between two pulses of an oscillator, a single increment of the central processing unit (CPU) clock during which the smallest unit of processor activity is carried out.

Definition 1.3.5. Latency is the time between the start of an operation and the moment its results become available or the number of clock cycles required to complete an operation. **Latency of a loop** is the number of clock cycles required to complete one iteration of the loop.

Definition 1.3.6. Throughput is the number of operations that can be completed in a given amount of time.

Definition 1.3.7. Initiation Interval (II) is the number of clock cycles between the start of two consecutive iterations of a loop. That is, it is the maximum rate (in clock cycles) at which loop iterations can be initiated. In the ideal case, the II is equal to 1 so that we can start a new iteration of the loop every clock cycle. Initiation interval is different from latency. The reason for this is pipelining which will be described in Section ??. For a visual explanation, see Figure 1.3.

Definition 1.3.8. Trip count is simply the number of iterations of a loop.



```
void top (a,b,c,d) {

  func_A(a,b,i1[N]);
  func_B(c,i1[N],i2[N]);
  func_C(i2[N],d)

}
```

```
void sub_func(...) {

 for (i=1;i>=0;i--) {
      op_Read;
      op_Compute;
      op_Write;
  }
}
```

Figure 1.3: Latency vs Initiation Interval illustration. Source: [2]

HLS synthesis

In this section, HLS synthesis will be described [15]. It is now the common workflow in the FPGA development because it significantly improves the productivity when working with design.

HLS synthesis is a methodology that bridges the gap between high-level programming languages, such as C, C++, or OpenCL, and the low-level hardware description languages typically used in FPGA design, like Verilog or VHDL. This approach enables developers to describe their algorithms and functionality at a higher level of abstraction, allowing them to focus on the problem-solving aspect rather than the intricacies of hardware implementation.

The HLS process starts with a high-level description of the desired functionality. This description can be written in familiar programming languages (in this case, C++ HLS), taking advantage of their abstractions and concise syntax. The HLS tool then performs a series of transformations on this high-level description to generate an optimized hardware implementation that can be deployed on an FPGA. The high-level description is transformed by the HLS tool into a RTL (Register Transfer Level) representation, which is the low-level hardware description that defines the behavior of the FPGA.

Simulation, Cosimulation

In this section, the processes of **simulation** and **cosimulation** will be described. In the context of developing for Field-Programmable Gate Arrays (FPGAs), simulation and cosimulation are two crucial techniques for verifying and testing the functionality of your design before actually programming it onto the FPGA hardware.

Definition 1.3.9. Simulation is the process of running a software-based model of your FPGA design on a computer to simulate its behavior. The process is very similar to running a software program on a computer for the purpose of unit testing certain parts of functionality of your code [16]. That is, the simulation does not involve any RTL code and is simply a software simulation of the high-level description..

Remark 1.3.10. Simulation might not always capture all aspects of hardware behavior, such as timing delays, which can be critical on FPGAs.

Definition 1.3.11. Cosimulation is a technique that combines simulation of the high-level description with simulation of the generated RTL description. This means that the simulations of both the original high-level code and the RTL representation in parallel, comparing their behavior. The purpose of cosimulation is to ensure that the high-level synthesis tool accurately transformed the high-level description into the desired RTL behavior. [16].

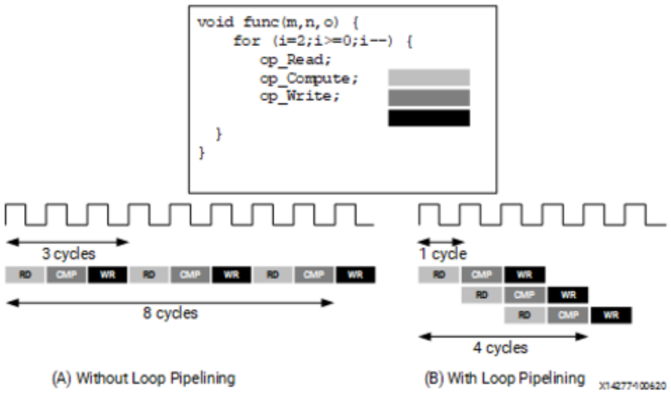
1.3.3 Common optimizations

In this section, common optimization techniques and how they are achieved will be introduced.

It is quite common to process data blocks (for example, a sequence of samples in anomaly detection) using for loops. For loops are usually the main bottleneck in the performance of the design and it is the area where most of the optimizations are applied first [15].

Loop Pipelining

Loop pipelining is a technique used in FPGAs programming to optimize the performance of sequential operations within a loop. It improves the throughput of loops by breaking them down into multiple stages that can execute concurrently. That is, it allows to start the next iteration of a loop before the current iteration has finished [3]. Refer to Figure 1.4 and Figure 1.3 for a visual illustration.



```
void func(m,n,o) {
    for (i=2;i>=0;i--) {
        op_Read;
        op_Compute;
        op_Write;

    }
}
```

Figure 1.4: Loop pipelining illustration. Source: [3]

Example 1.3.1. *Consider example of pipelining a simple for loop which adds 2 vectors ([3]):*

```
void toplevel(int* a, int* b, int* c, int len) {
  vadd: for(int i = 0; i < len; i++) {
#pragma HLS PIPELINE
      c[i] = a[i] + b[i];
  }
}
```

Taking the reference numbers from [3], assume that len is 20 and that one loop iteration takes 3 clock cycles, then the total latency of the loop is 60 clock cycles without pipelining.

The pipelining pragma `#pragma HLS PIPELINE` *allows to start the next iteration of the loop before the current iteration has finished.*

By default, the pipelining pragma will try to achieve an II of 1 but this can be specified manually by using the `II` *parameter.*

So, the pipelining pragma reduces the latency of the loop to 22 clock cycles.

In general, the latency of a loop with pipelining is given by the following formula:

$$Latency = II \cdot (Trip\ Count - 1) + Loop\ Body\ Latency \tag{1.3.1}$$

Remark 1.3.12. It is not always possible to achieve an II of 1. This is because of the dependencies between the iterations of the loop. For example, if the loop body depends on the result of the previous iteration, then the II cannot be 1 and we have to wait until the previous iteration has finished before starting the next one.

Loop Unrolling

Loop unrolling is an optimization technique that involves expanding or unwinding loops in code to potentially enable better utilization of hardware resources and/or to minimize control flow (branching) in loop iterations. This technique is not limited to FPGA development but can be particularly useful in optimizing code for FPGA implementations.

Loop unrolling works by duplicating loop iterations (read as `copy-pasting`). This allows utilizing more hardware as the loop body is duplicated multiple times and loop iterations will utilize different hardware resources. This increase in performance (i.e., throughput) comes at the cost of increased resource utilization [3].

Example 1.3.2. *Consider a simple example of a function which multiplies the input vector of length 4 by a constant, 2:*

```
void toplevel(int* a, int* b) {
  smult: for(int i = 0; i < 4; i++) {
#pragma UNROLL
      b[i] = 2 * a[i];
  }
}
```

The unroll pragma `#pragma UNROLL` *allows to unroll the loop and execute the loop iterations in parallel.*

The unrolled version of the loop will be equivalent to the following code:

```
void toplevel(int* a, int* b) {
  b[0] = 2 * a[0];
  b[1] = 2 * a[1];
  b[2] = 2 * a[2];
  b[3] = 2 * a[3];
}
```

Remark 1.3.13. It might not be possible to unroll a loop. For example, if the trip count is not known at compile time then the loop cannot be unrolled. Sometimes it is possible to unroll a loop partially. The unroll pragma allows to specify the factor of unrolling, i.e., how many iterations to unroll. For example, `#pragma UNROLL factor=2` will only duplicate the loop body so that there are 2 of them.

Loop Reordering

Loop reordering optimization is an optimization used to improve the performance by changing the order in which loops are executed. This optimization is not strictly related to FPGA development and it involves altering the nesting order of loops in a way that improves data locality and cache utilization (for example, on modern CPU), enables usage of SIMD resources.

In the context of FPGA, loop reordering can be used achieve better `II` when we are dealing with pipelining and nested loops (see Section 1.3.3 and Section 1.3.4 for an example).

Function Inlining

Function inlining optimization technique is not specific to FPGA development only. The technique is used to improve the performance of a program by reducing the overhead associated with function calls. Calling a function incurs some overhead in terms of memory and execution time due to the need to set up the function call stack, pass arguments, and jump to the function's code. Function inlining aims to eliminate this overhead by replacing a function call with the actual body of the function at the call site. In other words, the compiler takes the contents of the called function and inserts it directly into the location where the function is called. This, however, increases the size of the code and, specifically in the case of FPGAs, the resource utilization (LUTs and FF).

Array Partitioning and Reshaping

Partitioning arrays in an FPGA (Field-Programmable Gate Array) refers to the process of dividing a large memory block (e.g., one array) into smaller sections, often referred to as memory banks or partitions.

Partitioning allows accessing different parts of the array in parallel so that bottlenecks caused by a single memory interface being overwhelmed with requests can be avoided.

There are 3 different types of partitioning that can be applied to an array (Refer to Figure 1.5):

1. **Cyclic**. In a cyclic partition, the array is divided blocks of interleaved elements of the original array.
2. **Block**. In a block partition, the array is divided into non-overlapping blocks of sequential elements in the original array. Each block is assigned to a separate memory bank.
3. **Complete**. The array is split intio intividual elements

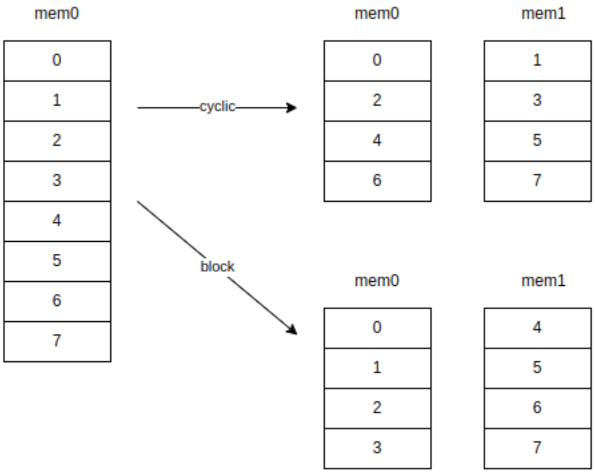


Figure 1.5: Array partitioning illustration.

The disadvantage of partitioning is that it increases the number of memory interfaces which leads to the increased resource utilization (i.e., more FFs, LUTs are used because each memory block requires separate control logic).

This can be partly mitigated by using the **array reshaping**. The difference between partitioning and reshaping is that partitioning creates multiple memory interfaces while reshaping still uses a single memory interface (i.e., all partitions are merged to a single physical memory).

1.3.4 Example of optimizing matrix multiplication

In this section, we will describe the process of optimizing a matrix multiplication using the techniques described in Section 1.3.3. This section can be treated as a tutorial on how to optimize a simple matrix multiplication.

Full source code for with all the files can be located in `vitis_hls/matmul_naive`.

Naive implementation

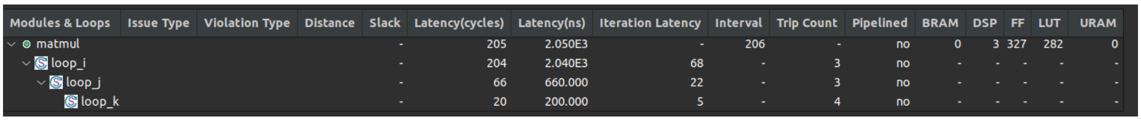
The naive implementation of matrix multiplication is simply a triple for loop which directly implements the definition of $C = A \cdot B$ where A, B and C are the matrices and $C_{i,j}$ are defined as in Equation 1.3.2.

$$C_{i,j} = \sum_{k=0}^{N} A_{i,k} \cdot B_{k,j} \tag{1.3.2}$$

```
#include "matrixmul.h"

void matmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
            mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
            result_t res[MAT_A_ROWS][MAT_B_COLS]) {
loop_i:
  for (int i = 0; i < MAT_A_ROWS; i++) {
  loop_j:
    for (int j = 0; j < MAT_B_COLS; j++) {
      res[i][j] = 0;
    loop_k:
      for (int k = 0; k < MAT_B_ROWS; k++) {
#pragma HLS PIPELINE off
        res[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

The Vitis HLS Synthesis Report for the naive implementation is presented in Figure 1.6. The data types for matrices used are int32_t. Matrix A is of size 3x4 and matrix B has size 4x3. For this baseline implementation, we are reaching a latency of 205 clock cycles, using 0 BRAMs, 327 FF and 282 LUTs.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmul				-	205	2.050E3	-	206	-	no	0	3	327	282	0
loop_i				-	204	2.040E3	68	-	3	no	-	-	-	-	-
loop_j				-	66	660.000	22	-	3	no	-	-	-	-	-
loop_k				-	20	200.000	5	-	4	no	-	-	-	-	-

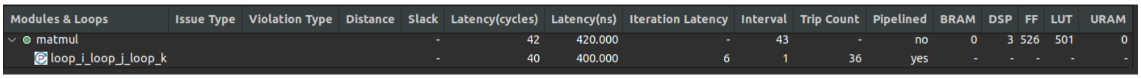
Figure 1.6: Naive matrix multiplication synthesis report

Loop pipelining and unrolling

The baseline code can be optimized by pipelining. There are three loops that can be pipelined (`loop_i`, `loop_j` and `loop_k`).

Case 1: Pipelining `loop_k`. A simple pipelining of the innermost loop leads to the pipelining of Multiply and Accumulate operation (MAC) which is the main operation in the loop. There is no need to partition arrays a and b as memory in arrays a and b only need to supply 1 element per cycle.

The pipelining leads to decrease in latency to 42 clock cycles at the cost of using 526 FF and 501 LUT which is almost twice as many resources as in the naive implementation.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmul				-	42	420.000	-	43	-	no	0	3	526	501	0
loop_i_loop_j_loop_k				-	40	400.000	6	1	36	yes	-	-	-	-	-

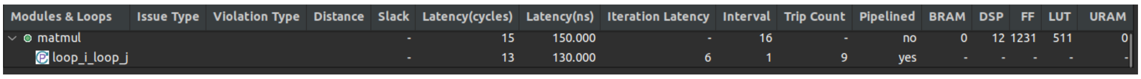
Figure 1.7: Matrix multiplication with pipelined loop `loop_k`

Case 2: Pipelining `loop_j`. The pipelining of the middle loop requires partitioning of arrays a by `MAT_A_COLS` and b by `MAT_B_ROWS`. There are `MAT_A_COLS` (or `MAT_B_ROWS`) MAC operations per cycle so that memory of a and b needs to be sufficiently divided to supply `MAT_B_ROWS` elements per cycle. Array a is partitioned along the second dimension and array b is partitioned along the first dimension because of the access patterns. In this example, we use the `complete` partitioning which divides the memory into individual registers.

```
#include "matrixmul.h"

void matmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
            mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
            result_t res[MAT_A_ROWS][MAT_B_COLS]) {
#pragma HLS ARRAY_PARTITION variable = a complete dim = 2
#pragma HLS ARRAY_PARTITION variable = b complete dim = 1
loop_i:
  for (int i = 0; i < MAT_A_ROWS; i++) {
#pragma HLS PIPELINE off
  loop_j:
    for (int j = 0; j < MAT_B_COLS; j++) {
      int tmp = 0;
#pragma HLS PIPELINE
    loop_k:
      for (int k = 0; k < MAT_B_ROWS; k++) {
#pragma HLS UNROLL
        tmp += a[i][k] * b[k][j];
      }
      res[i][j] = tmp;
    }
  }
}
```

With this addition, the latency is down to 15 clock cycles and the resource usage has increased to 1221 FF and 511 LUTs (Figure 1.8).



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmul				-	15	150.000	-	16	-	no	0	12	1231	511	0
loop_i_loop_j				-	13	130.000	6	1	9	yes	-	-	-	-	-

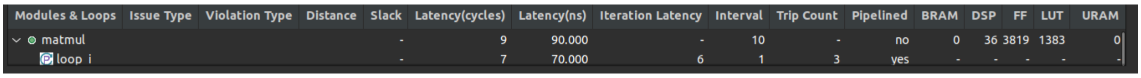
Figure 1.8: Matrix multiplication with pipelined loop `loop_j`

Case 3: Pipelining `loop_i`. Pipelining the outermost loop requires additional partitioning of array res by `MAT_B_COLS` and full partitioning of array b as `loop_k` and `loop_j` are getting unrolled.

```
#include "matrixmul.h"

void matmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
            mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
            result_t res[MAT_A_ROWS][MAT_B_COLS]) {
#pragma HLS ARRAY_PARTITION variable = a complete dim = 2
#pragma HLS ARRAY_PARTITION variable = b complete dim = 0
#pragma HLS ARRAY_PARTITION variable = res complete dim = 2
loop_i:
  for (int i = 0; i < MAT_A_ROWS; i++) {
#pragma HLS PIPELINE
  loop_j:
    for (int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS UNROLL
      int tmp = 0;
    loop_k:
      for (int k = 0; k < MAT_B_ROWS; k++) {
#pragma HLS UNROLL
        tmp += a[i][k] * b[k][j];
      }
      res[i][j] = tmp;
    }
  }
}
```

The latency is down to 9 clock cycles and the resource usage is up to 3819 FF and 1383 LUTs (Figure 1.9).



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmul				-	9	90.000	-	10	-	no	0	36	3819	1383	0
loop_i				-	7	70.000	6	1	3	yes	-	-	-	-	-

Figure 1.9: Matrix multiplication with pipelined loop `loop_i`

Loop reordering

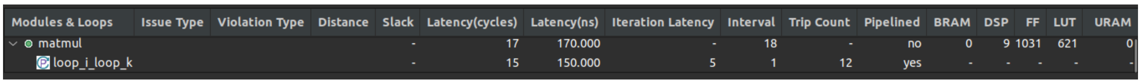
We can also compare how the loop reordering affects the performance of the matrix multiplication and specifically how it compares to the pipelining of the `loop_j` in terms of latency and resource usage.

The loop reordering technique allows us to avoid partitioning matrix a compared to the `loop_j` pipelining solution. However, we now have to partition the output matrix by `MAT_B_COLS`.

```
#include "matrixmul.h"

void matmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
            mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
            result_t res[MAT_A_ROWS][MAT_B_COLS]) {
  int temp_sum[MAT_B_COLS];
#pragma HLS ARRAY_PARTITION variable = b dim = 2 complete
#pragma HLS ARRAY_PARTITION variable = res dim = 2 complete
#pragma HLS ARRAY_PARTITION variable = temp_sum dim = 1 complete
loop_i:
  for (int i = 0; i < MAT_A_ROWS; i++) {
  loop_k:
    for (int k = 0; k < MAT_B_ROWS; k++) {
#pragma HLS PIPELINE
    loop_j:
      for (int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS UNROLL
        int result = (k == 0) ? 0 : temp_sum[j];
        result += a[i][k] * b[k][j];
        temp_sum[j] = result;
        if (k == MAT_B_ROWS - 1) {
          res[i][j] = result;
        }
      }
    }
  }
}
```

From Figure 1.10, the latency is 17 clock cycles and the resource usage is 1031 FF and 611 LUTs. The performance is worse than the pipelined loop `loop_j` but the resource usage is lower for FF and higher for LUTs.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmul				-	17	170.000	-	18	-	no	0	9	1031	621	0
loop_i_loop_k				-	15	150.000	5	1	12	yes	-	-	-	-	-

Figure 1.10: Matrix multiplication with loop reordering

Summary The pipelining of the outermost loop leads to the best performance with the lowest latency. However, this comes at the big cost of using the most resources as array b needs to be fully partitioned. In this work, we will use the pipelined loop `loop_j` as it provides a good trade-off between latency and resource usage.

Chapter 2

Experiments

2.1 Architecture and hyperparameters

Here we will describe the model architecture and the hyperparameters used for the experiments.

In the experiments 3 models were used for comparison:

- **Linear Regression** - a simple linear regression model on handcrafted features
- **Transformer Encoder** - a transformer encoder model on raw time series data
- **Linear Transformer Encoder** - a linear transformer model on raw time series data

For both transformer models, we used the encoder architectures as described in Section 1.2.1 with a linear layer on top of the output of the transformer encoder to get the final prediction (i.e., if a sample is an anomaly).

Remark 2.1.1. The encoder part has positional encoding and layer normalization disabled. We found that the positional encoding does not improve the performance of the model and leads to more unstable training (see Section 2.1.1).

The transformer hyperparameters used for the experiments are presented in Table 2.1.

Parameter, Model	Transformer Encoder	Linear Transformer Encoder
Window Size	8	8
Number of heads	8	-
Dim. of FeedForward network	16	16
Number of blocks	2	1

Table 2.1: Hyperparameters

The learning rate was chosen using the learning rate finder [17] (see Section 2.1.1) and the batch size was chosen to be the maximum value that fitted the dataset in memory (2^{13} samples).

2.1.1 Model Fitting

In this section, we will describe the training procedure and the main issues that we encountered and how they were addressed.

General procedure

The General process of training a neural network involves iteratively adjusting its parameters to minimize a specified loss function. This is typically achieved through an optimization algorithm, such as gradient descent.

That is, at each iteration, the parameters of the model θ are updated as follows:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t) \tag{2.1.1}$$

where α is the learning rate and $\mathcal{L}(\theta_t)$ is the loss function at the t-th iteration.

While the general procedure is simple there are multiple methods to improve the convergence of the optimization algorithm, which will be described in the following sections.

Learning rate finder

The learning rate α is one of the most important hyperparameters that determines the step size in parameter space during gradient descent optimization. An appropriate learning rate is essential for model convergence. The problem of choosing the learning rate is a well-known problem in machine learning and badly chosen learning rate can lead to either underfitting where the model learns too slowly or it can lead to divergence where the parameters are updated too abruptly.

In [17], authors proposed a simple method to find an appropriate learning rate automatically by plotting the loss function against the learning rate.

The procedure is performed as follows:

1. Start with a very small learning rate α and increase it at each iteration

2. At each iteration, train the model for a few epochs and compute the loss function

3. Plot the loss function against the learning rate This plot is crucial in identifying the ”sweet spot” in the learning rate range where the loss is decreasing effectively.

4. Choose the point on the learning rate vs. loss curve where the loss starts to decrease most steeply. This point indicates that the model is making the most significant progress towards convergence.

Remark 2.1.2. While there are no guarantees that the learning rate finder will find the optimal learning rate, it provides a good empirical estimate of the optimal learning rate.

Instead of manually plotting the loss function against the learning rate, we used the implementation provided in Pytorch Lightning [18].

Gradient explosion and Gradient clipping

A different challenge of training neural networks is the phenomenon known as the ”gradient explosion problem.” We have found that the gradient explosion problem is especially prevalent in the proposed architectures.

The gradient explosion problem occurs when the gradient of the loss function with respect to the parameters becomes too large and the parameters are updated too abruptly (e.g., as in Equation 2.1.1). This can lead to the model diverging and the loss function increasing instead of decreasing. An example of the loss function diverging is presented in Figure 2.1.

To solve the problem of gradient explosion, we used the gradient clipping technique introduced in [19]. The idea behind gradient clipping is to clip the gradient to a maximum value g_{max}. This remediates the problem of gradient explosion because the gradient is bounded and the parameters are updated more smoothly.

Class imbalance and loss functions

In anomaly detection task the dataset is often imbalanced, i.e., the number of normal samples is much larger than the number of anomalous samples.

This poses a problem for the training of the model because the model can simply learn to predict the majority class (i.e., normal samples) and achieve a high accuracy without having good recall (refer to Section 2.3.2 for the description to the metrics).

The loss functions that we use for training is the binary cross-entropy loss function [20] which is computed as follows:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{2.1.2}$$

where y_i is the true label and $\hat{y}_i$ is the predicted label.

While the binary cross-entropy loss function is a good choice for the anomaly detection task, it does not take into account the class imbalance problem.

A way to solve the class imbalance problem is to use a weigh positive samples (i.e., anomalous samples) more than negative samples in the loss function.

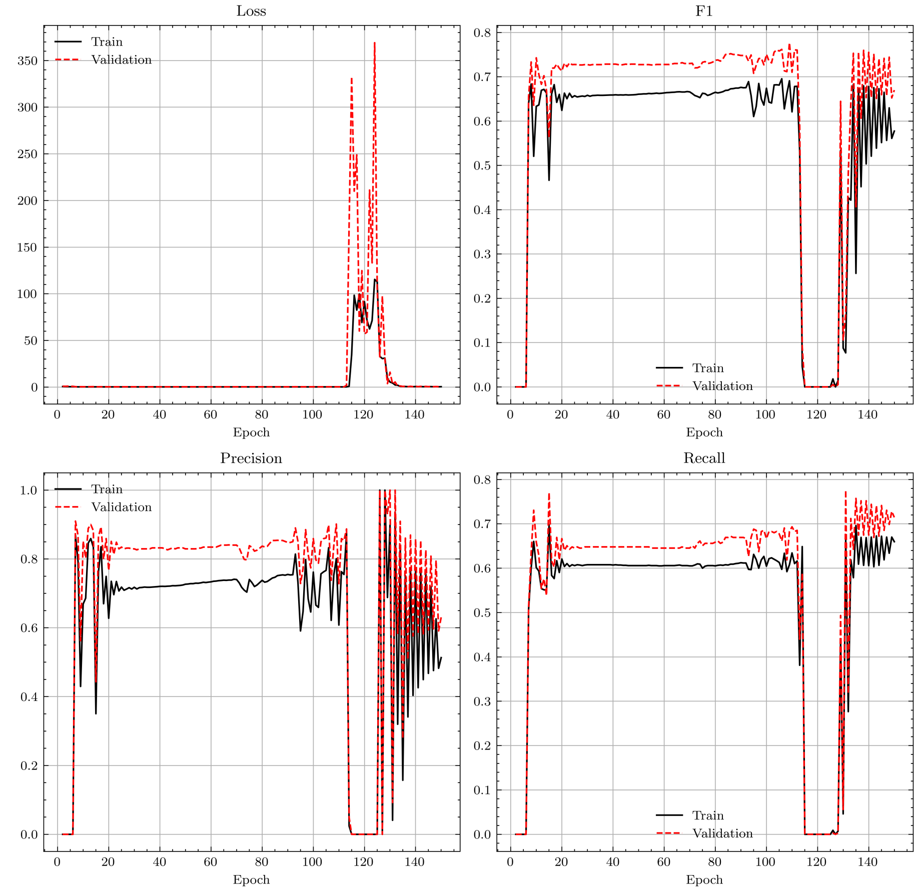


Figure 2.1: Example of gradient explosion at around epoch 115 which leads to the loss function diverging for a few following epochs.

So we can modify the loss function as follows:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} w_i (y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \tag{2.1.3}$$

where w_i is the weight of the i-th sample. In the experiments, we found that weighting the positive samples 5 times more than the negative samples yielded good results for most of the datasets which we used throughout the experiments.

Training stability: Layer normalization and positional encoding

While the base transformer encoder architecture uses layer normalization and positional encoding layers, we found that they lead to unstable training and worse performance of the model on most of the datasets. Hence, we disabled them for the experiments and replaced them with identity layers.

2.2 Datasets

In this section, the datasets used for model training and performance evaluation will be described.

2.2.1 Numenta Anomaly Benchmark (NAB)

To assess the accuracy of predictions, we use the Numenta Anomaly Benchmark [21] dataset, which contains various real-world labeled time series of temperature sensor readings, CPU utilization of cloud machines, service request latencies, and taxi demands in New York City. It is commonly used to assess the performance of anomaly detection models on time-series data.

The reason why we use this dataset is that it is a standard benchmark dataset for anomaly detection in time series and because it has a large number of labeled time series.

A sample time series of NYC taxi demand is presented in Figure 2.2.

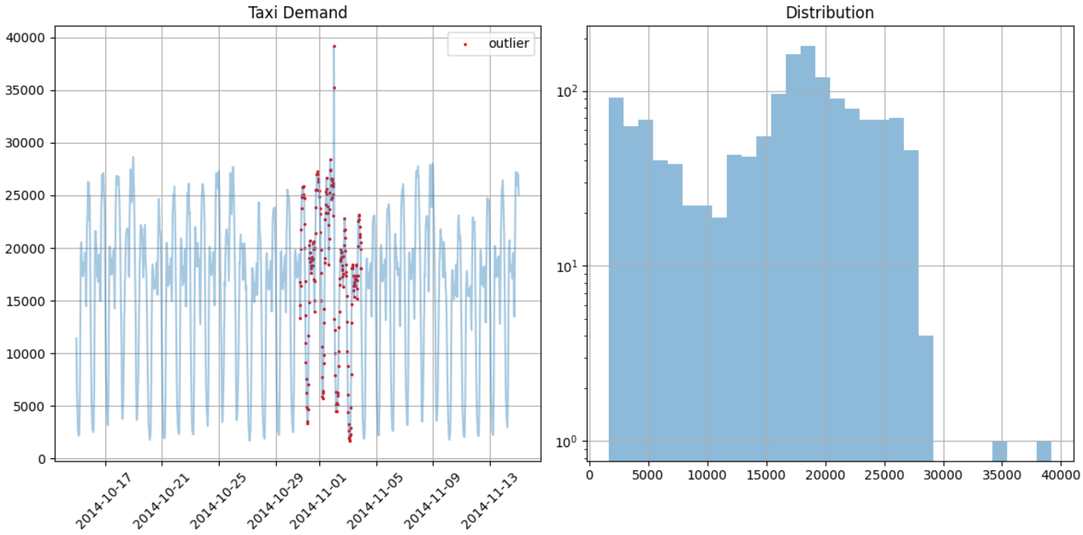


Figure 2.2: NYC Taxi demand - anomalies highlighted in red

2.2.2 KPI Anomaly Detection Dataset

The other labeled dataset that we use is the KPI Anomaly Detection Dataset (KPI AIOps) [22]. This dataset alongside the NAB dataset will be used to evaluate the predictive performance of the anomaly detection models.

The dataset consists of KPI (key performace index) time series data from many real scenarios of Internet companies with ground truth label. KPIs fall into two broad categories: service KPIs and machine KPIs. Service KPIs are performance metrics that reflect the size and quality of a Web service, such as page response time, page views, and number of connection errors. Machine KPIs are performance indicators that reflect the health of the machine (server, router, switch), such as CPU utilization, memory utilization, disk IO and network card throughput.

A sample time series of a sensor readings is presented in Figure 2.2. We can clearly see the outliers for some of the observations (colored in red).

2.2.3 FI2010

In [23], authors described the first publicly available benchmark dataset of high-frequency limit order markets for mid-price prediction. The dataset contains 10-day limit order book data from June 2010 for five stocks that are listed on the Helsinki exchange. Each entry in the time series provides price details and aggregate order sizes for the top ten levels on both the bid and offer sides of the market, totaling forty data points. The time series consists of approximately four million messages, representing incoming buy/sell orders or cancellations. The dataset features order book snapshots taken after every 10 messages, resulting in approximately 400,000 records for the five stocks.

A number of versions of the dataset are available using different normalization schemes. We used the not normalized version of the dataset.

For the purpose of this work, we only extract only the mid price from the dataset which will be used for anomaly detection task.

Synthetic outliers

Since the dataset is not labeled, we have to inject synthetic anomalies into the dataset. We employ the approach similar to [24] with a slight modification. The algorithm can be summarized as follows:

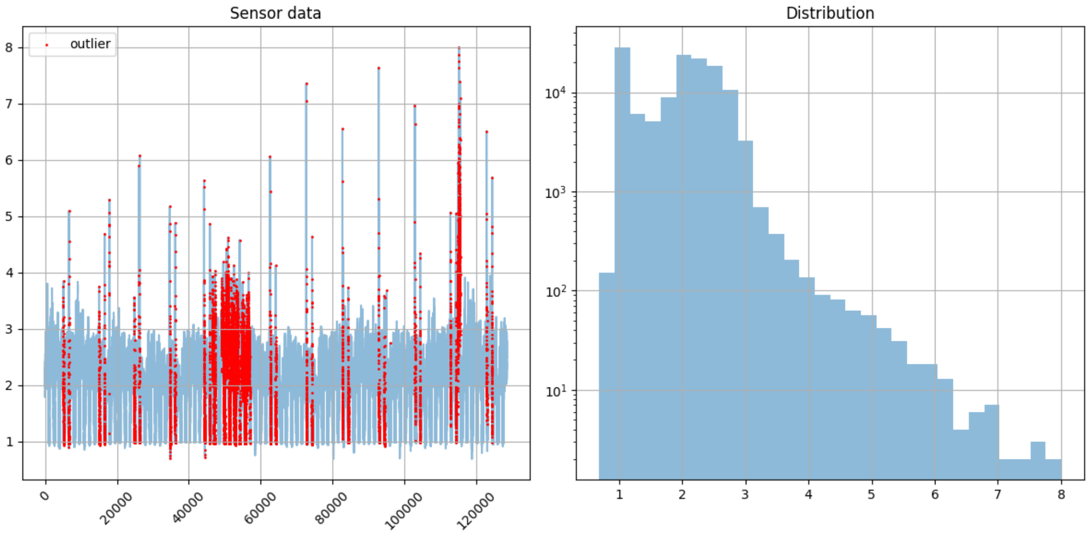


Figure 2.3: Sensor data from a machine in a data center. The red dots indicate the anomalies.

1. Select n samples from the time series which will be contaminated (i.e., anomalous)
2. Replace the sample S_i with $\hat{S}_i = S_i(1+\delta)$ where δ is the injected outlier in the return space.

Authors model δ as a uniformly distributed random variable $\mathcal{U}[0, \rho]$. We instead use the normal distribution with matching mean and standard deviation of the returns time series.

An example of the injected outliers is presented in Figure 2.4

2.3 Accuracy

In this section, we will describe the main metrics used to evaluate the performance of the anomaly detection models. The Transformer encoder model will be compared with the simple linear regression model on handcrafted features and with the Linear Transformer model [14]. The inference procedure will be described and the results will be presented.

2.3.1 Inference

After training the model on the training set as described in Section 2.1.1, we can use the model to make predictions on the test set.

2.3.2 Metrics

In this section the metrics used to evaluate the performance of the anomaly detection models will be described.

Before we describe the metrics, we need to introduce the confusion matrix and the following notation:

- **TP** - True Positive calculated as the number of correctly predicted anomalies
- **TN** - True Negative calculated as the number of correctly predicted non-anomalies
- **FP** - False Positive which is the number of incorrectly predicted anomalies
- **FN** - False Negative which is the number of incorrectly predicted non-anomalies
- **P** - Number of positive samples, i.e., $\mathbf{P} = \mathbf{TP} + \mathbf{FN}$
- **N** - Number of negative samples, i.e., $\mathbf{N} = \mathbf{TN} + \mathbf{FP}$

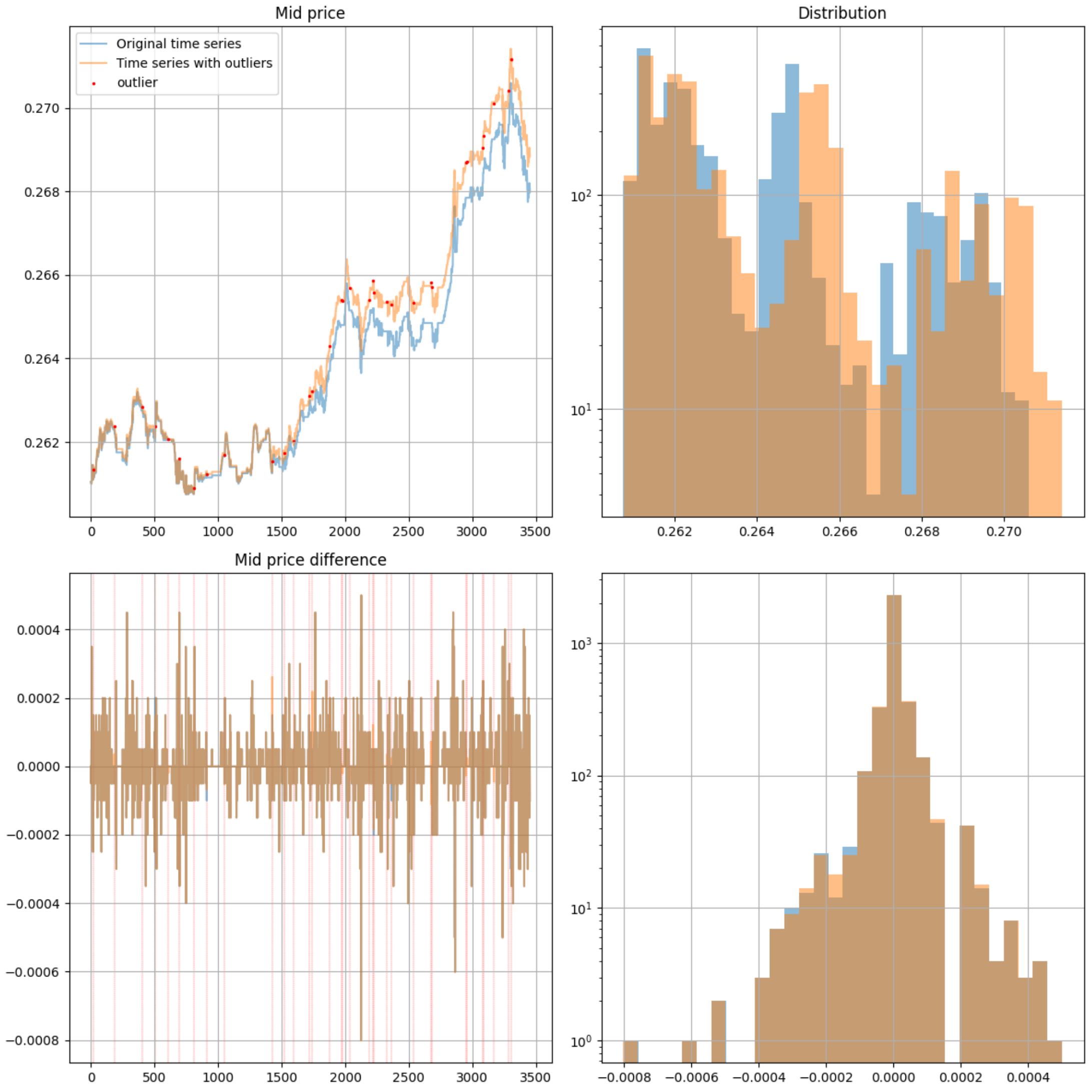


Figure 2.4: Example of the injected outliers in the FI2010 dataset.

The confusion matrix is a table with two rows and two columns that reports the number of false positives, false negatives, true positives, and true negatives.

prediction outcome

		P	**n**	**total**
actual value	**P′**	True positive **TP**	False negative **FN**	P′
	N′	False positive **FP**	True negative **TN**	N′
	total	P	N	

The matrix summarizes the predictions from a classification model, i.e., how well the model performed when predicting the class labels for positive and negative samples. While the matrix presents the most informative view of the performance of the model, we still need to summarize the information in the matrix into a single number(s) that can be used to compare different models.

In this paper, we will use the following metrics to compare the performance of the anomaly detection models:

- **Accuracy** is the fraction of predictions that the model got right. It is defined as follows:

  $$\text{Accuracy} = \frac{\mathbf{TP} + \mathbf{TN}}{\mathbf{TP} + \mathbf{TN} + \mathbf{FP} + \mathbf{FN}}$$

  While this metric is easy to understand, it is not very informative when the dataset is imbalanced which is the case for the anomaly detection task where the proportion of positive samples is low.

- **Precision** is the fraction of positive predictions that were correct.

  $$\text{Precision} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FP}}$$

  This metric is useful when the cost of false positives is high. For example, in the case of anomaly detection, we want to have a high precision so that we do not have to manually check many false positives or trigger any downstream filtering task too often.

- **Recall** is the fraction of positive samples that were correctly predicted.

  $$\text{Recall} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FN}}$$

  This metric is useful when the cost of false negatives is high. For example, in the case of anomaly detection, we want to minimize the number of false negatives because we do not want to miss any anomalies.

- **F1** is the harmonic mean of precision and recall which ranges from 0 to 1.

  $$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

  This metric is useful when we want to balance the precision and recall. For example, in the case of anomaly detection, we want to have a high precision so that we do not have to manually check many false positives or trigger any downstream filtering task too often. At the same time, we want to minimize the number of false negatives because we do not want to miss any anomalies.

  The advantage of using this metric instead of the accuracy is that it can be used even when the dataset is highly imbalanced and it would detect if the model performs poorly in terms of precision and/or recall.

2.3.3 Comparison

2.4 Performance/Speed

2.5 Resource utilization on FPGA

Conclusion

2.6 Future work

Bigger FPGA boards.

Evaluation of performance on more recent financial market data.

Appendix A

Code

A.1 Efficient matrix multiplication

This is Appendix A.1, which usually contained supporting material, or complicated proofs that might make the main text above less readable / fluid.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[2] University of York. Vitis HLS Knowledge Base - Real-Time Systems - York Wiki Service, July 2020.

[3] Xilinx Inc. Vitis hls: Pipelining loops. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Design-Principles, May 2023.

[4] Thomas Neil Falkenberry CFA. High frequency data filtering, Sep 2008.

[5] Owen Vallis, Jordan Hochenbaum, and Twitter. Introducing practical and robust anomaly detection in a time series.

[6] Mohiuddin Ahmed, Nazim Choudhury, and Shahadat Uddin. Anomaly detection on big data in financial markets. In *2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 998–1001, 2017.

[7] Anthony Gillioz, Jacky Casas, Elena Mugellini, and Omar Abou Khaled. Overview of the transformer-based models for nlp tasks. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pages 179–183, 2020.

[8] Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. Transformers in time series: A survey. 2022.

[9] Andreea-Ingrid Funie, Liucheng Guo, Xinyu Niu, Wayne Luk, and Mark Salmon. Custom framework for run-time trading strategies. In Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro, editors, *Applied Reconfigurable Computing*, pages 154–167, Cham, 2017. Springer International Publishing.

[10] Julio-Omar Palacio-Niño and Fernando Berzal. Evaluation metrics for unsupervised learning algorithms, 2019.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[13] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[14] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020.

[15] Xilinx Inc. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Design-Principles, May 2023.

[16] Xilinx Inc. C/RTL Co-Simulation in Vitis HLS, July 2023.

[17] Leslie N. Smith. Cyclical learning rates for training neural networks, 2015.

[18] lightning.ai. Learningratefinder — pytorch lightning 2.0.7 documentation, Aug 2023.

[19] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2012.

[20] I. J. Good. Rational decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1):107–114, Jan 1952.

[21] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 11 2017. [Online; accessed 2023-07-19].

[22] Zeyan Li, Nengwen Zhao, Shenglin Zhang, Yongqian Sun, Pengfei Chen, Xidao Wen, Minghua Ma, and Dan Pei. Constructing large-scale real-world benchmark datasets for aiops, 2022.

[23] Adamantios Ntakaris, Martin Magris, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Benchmark dataset for mid-price forecasting of limit order book data with machine learning methods. 2017.

[24] Stéphane Crépey, Noureddine Lehdili, Nisrine Madhar, and Maud Thomas. Anomaly Detection in Financial Time Series by Principal Component Analysis and Neural Networks. *Algorithms*, 15(10):385, oct 19 2022. [Online; accessed 2023-07-19].