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 [2] [3]. Moreover, the amount of financial time-series data has been significantly increasing [4]. 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 [5]. 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 [6].

Increasingly powerful hardware, such as field-programmable gate arrays (FPGAs), have seen increasing usage in recent years due to their reconfigurability and high performance [7].

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 [8].

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 [9].

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

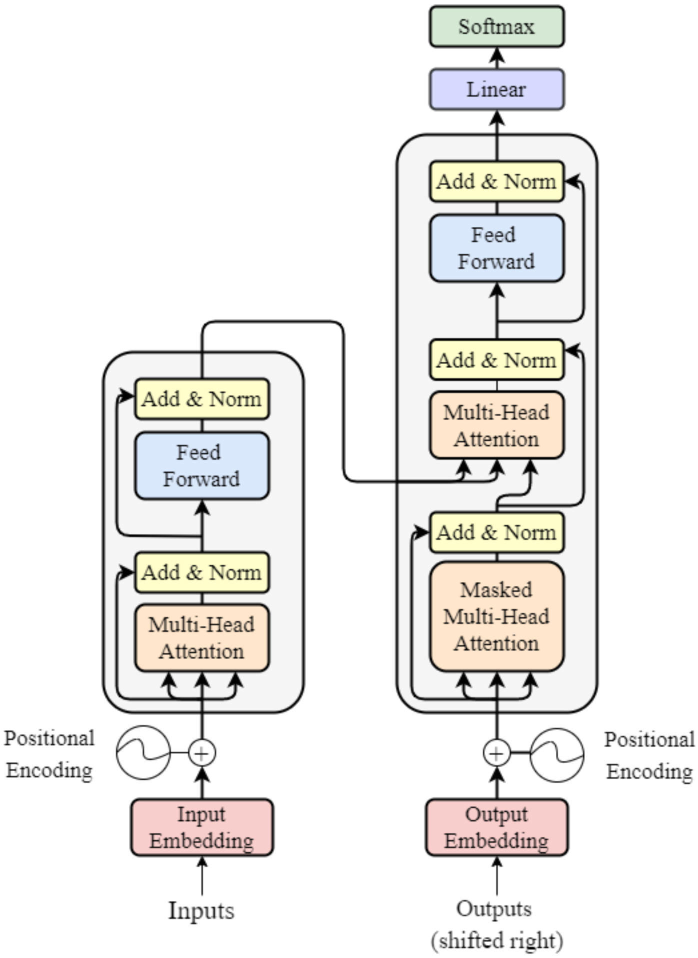

Figure 1.1: Model architecture of the Transformer [1]

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

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

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

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

The **Add & Norm** layer is a residual connection [10] followed by a layer normalization layers [11]. 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 $\mathrm{FC}(x)$ is simply a linear transformation inputs X (i.e., a matrix multiplication) with the activation function applied element-wise to the result. That is, $\mathrm{FC}(x) = f(W \cdot x + b)$ 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].

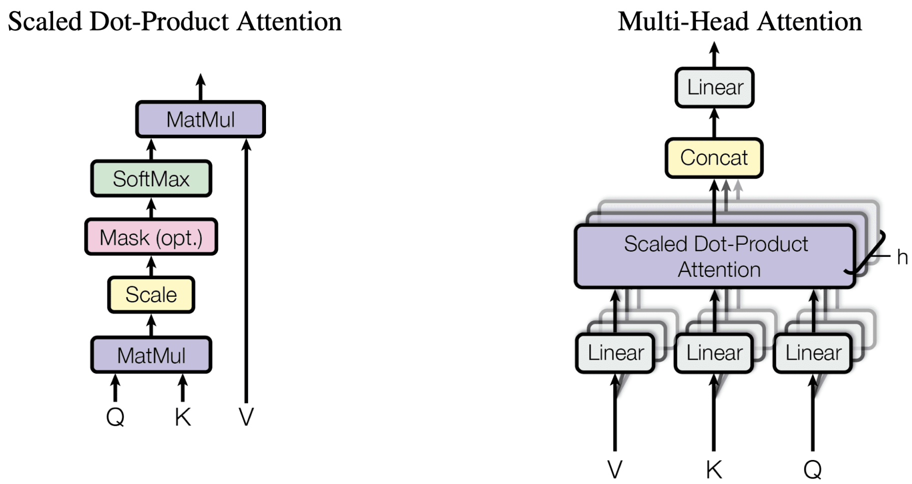

Figure 1.2: Scaled Dot-Product Attention and Multi-Head Attention [1]

Linear attention

In [12], 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 [12], 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 [7]. 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 and HLS

Common Terms

Simulation, Cosimulation

A way to design and debug the solution without running it on the board.

HLS synthesis

In this section, HLS synthesis will be described [13]. It is now the common workflow in the FPGA development because it significantly improves the productivity when working with design.

1.3.3 Common optimizations

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

Pipelining

Example code:

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

Loop Unrolling

Arrays

TODO: Partitioning

Streams

TODO:

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 [14] (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 [14], 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 [15].

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 [16]. 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 [17] 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.

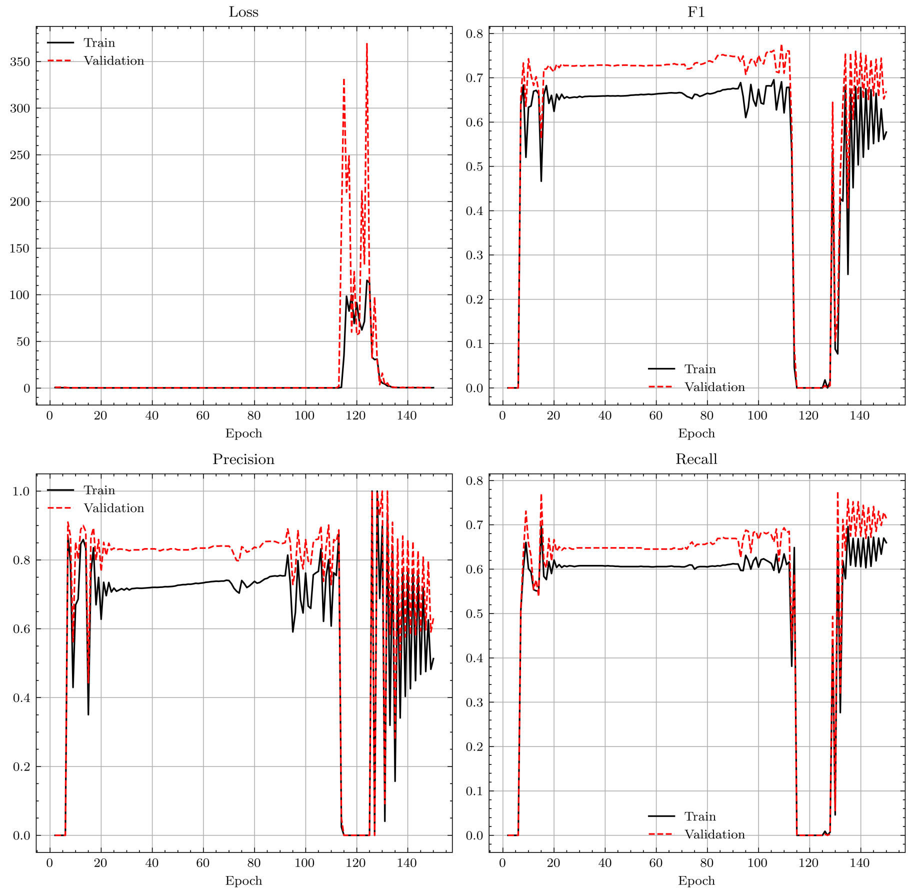

Figure 2.1: Example of gradient explosion at around epoch 115 which leads to the loss function diverging for a few following epochs.

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 [18] 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.

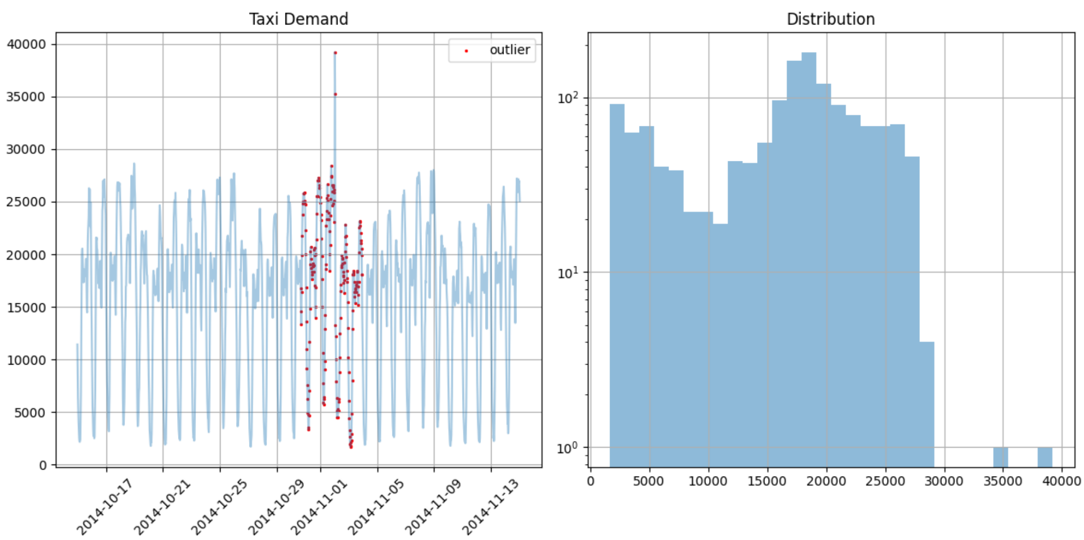

Figure 2.2: NYC Taxi demand - anomalies highlighted in red

2.2.2 KPI Anomaly Detection Dataset

The other labeled dataset that we use is the KPI Anomaly Detection Dataset (KPI AIOps) [19]. 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 [20], 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 [21] with a slight modification. The algorithm can be summarized as follows:

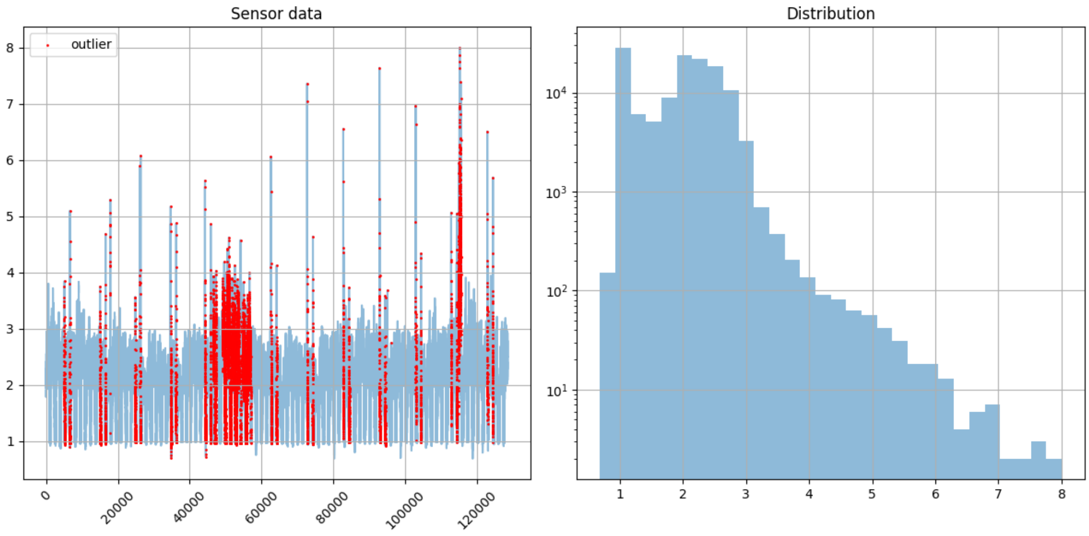

Figure 2.3: Sensor data from a machine in a data center. The red dots indicate the anomalies.

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 [12]. 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., $\mathbf{N} = \mathbf{TN} + \mathbf{FP}$

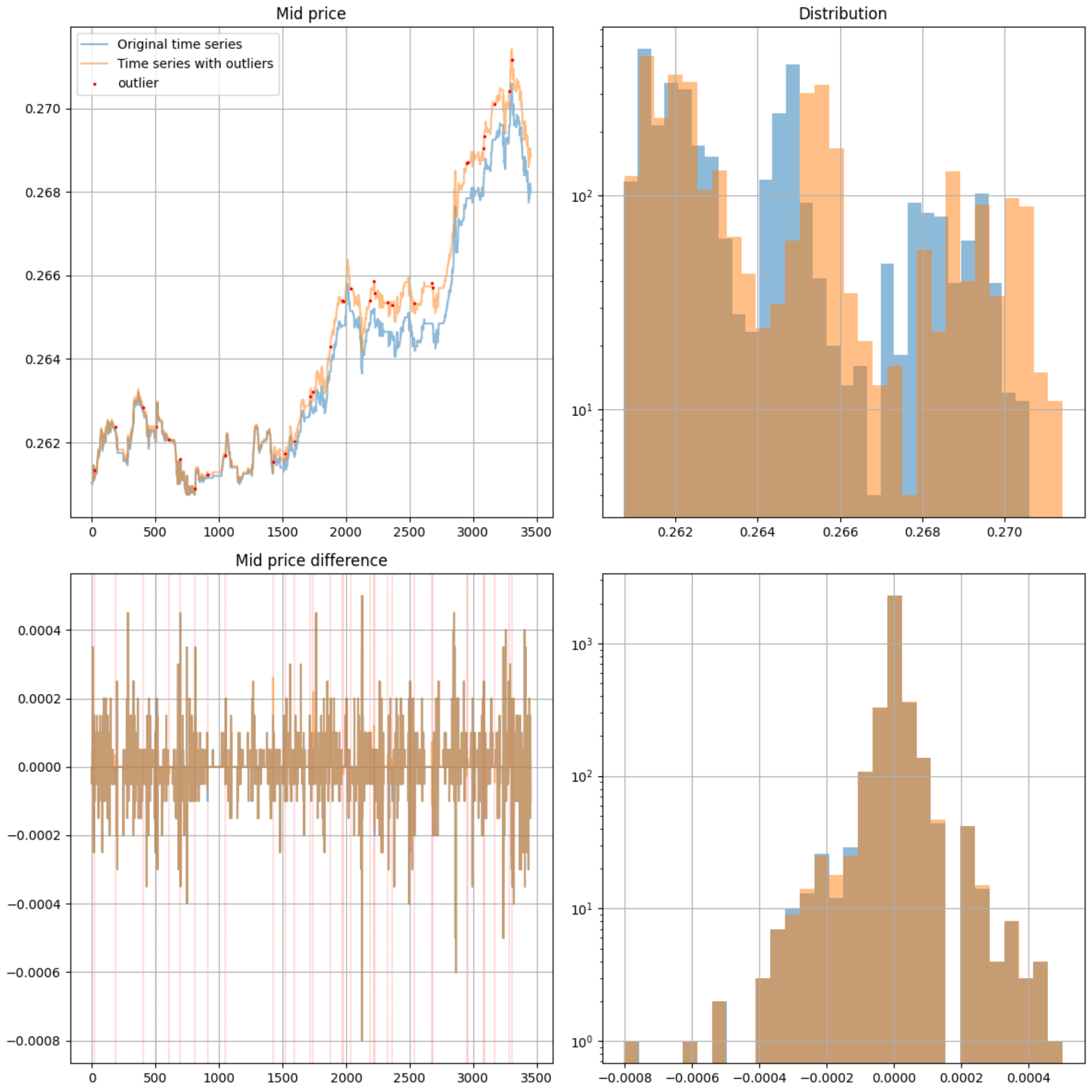

Figure 2.4: Example of the injected outliers in the FI2010 dataset.

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:

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

  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.

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

  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.

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

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

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

[13] Xilixn Inc. Vitis High-Level Synthesis User Guide, may 10 2023. [Online; accessed 2023-07-16].

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

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

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

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

[18] 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].

[19] 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.

[20] 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.

[21] 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].