Accelerating Machine Learning for Trading Using Programmable Switches

Xinpeng Hong[a,*], **Changgang Zheng**[a], **Stefan Zohren**[a] **and Noa Zilberman**[a]

[a]Department of Engineering Science, University of Oxford, Oxford, United Kingdom

Abstract. High-frequency trading (HFT) employs cutting-edge hardware for rapid decision-making and order execution but often relies on simpler algorithms that may miss deeper market trends. Conversely, lower-frequency algorithmic trading uses machine learning (ML) for better market predictions but higher latency can negate its strategic benefits. To achieve the best of both worlds, we present an in-network ML solution that embeds ML processes into programmable network devices, accelerating feature engineering and extraction as well as ML inference. In this paper, we design and develop a solution that supports both stock mid-price and volatility movement forecasting using commodity switches. Our approach achieves microsecond-scale, ultra-low latency, significantly lowering it by 64% to 97% compared to previous works, while upholding the same level of ML performance as server models. Additionally, by combining network hardware and servers, a hybrid deployment strategy can keep the misclassification rate change below 0.8% relative to the server baseline while processing 49% of the traffic directly on the switch and achieving a 45% average reduction in end-to-end latency.

1 Introduction

High-frequency trading (HFT), a form of algorithmic trading that distinguishes itself by extremely high speeds and microsecond-scale data processing, often leverages specialized hardware for acceleration [22]. In contrast, trading strategies that operate at lower frequencies typically rely on machine learning (ML) to improve decision-making processes by analyzing market conditions more accurately and thoroughly [19]. While combining the analytical and predictive power of ML with the rapid execution capabilities of HFT proves to be highly effective [24], merging ML-based strategic trading with HFT often leads to increased latency and a slowdown in trading speeds [11]. Recognizing the complementary strengths and weaknesses of HFT and ML-driven trading, this study focuses on bridging the gap between these two paradigms.

Application workloads can be partially or fully offloaded from end hosts to the network infrastructure, using programmable network devices that are already deployed for traffic forwarding. This is referred to as in-network computing [47]. Within this domain, in-network ML focuses on deploying pre-trained ML models directly within networking hardware to perform inference. The primary objectives of this strategy include achieving lower latency, higher throughput, and improved power efficiency [39, 49]. Figure 1 illustrates the difference in data paths between general ML and in-network ML. Functioning within a programmable switch, in-network ML can reduce latency by circumventing both end hosts and intermediate network devices between end hosts and the in-network ML node. Therefore, by design, in-network ML offers an effective solution for accelerating ML-driven applications in scenarios where latency is crucial, especially in trading activities.

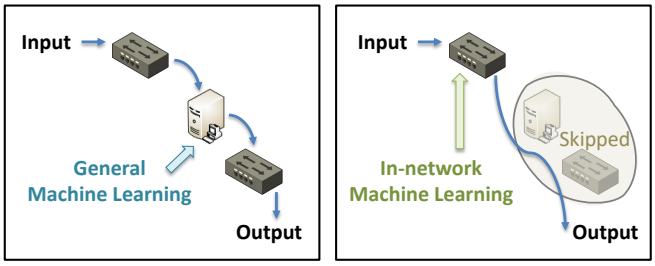


Figure 1: Data path difference between general ML and in-network ML.

Additionally, concerns have risen about the escalating burden on CPU cycles and excessive electrical power use driven by the competition among HFT firms [42]. Beyond mere acceleration, the inherent energy efficiency of network devices can provide $\times 1000$ power saving [46, 47], offering the potential to replace numerous data center servers and accelerators, thus enhancing AI sustainability.

Prior research on applying in-network ML for time-critical applications has been limited, with only a few works demonstrating its potential for tackling trading challenges [52, 55, 20, 21]. However, these studies were confined to stock price movement prediction and relied solely on raw features from market-by-order (MBO) or limit order book (LOB) data. MBO data consists of basic information about individual orders such as price and quantity [51], whereas LOB data aggregates these orders, displaying total buy and sell quantities at different price levels to reveal supply and demand dynamics [50]. Engineering and extracting richer features yields deeper market insights and enhanced ML outcomes, yet, to the best of our knowledge, this strategy has not been implemented within network devices before. To overcome the limitations of prior in-network ML research, this work emphasizes advanced feature engineering and extraction beyond raw data, alongside ML inference, at the network edge. As Figure 2 shows, this positioning allows for immediate prediction and decision-making, effectively bypassing the latency introduced by traditional processing flow and server-based inference. Our solution is evaluated on stock price movement prediction and, for the first time, on forecasting short-term stock volatility, which is another essential trading challenge. Tested on realistic datasets within programmable switches using diverse in-network ML models, our approach enhances the feasibility, practicality, and applicability of in-network ML.

* Corresponding Author. Email: xinpeng.hong@eng.ox.ac.uk.

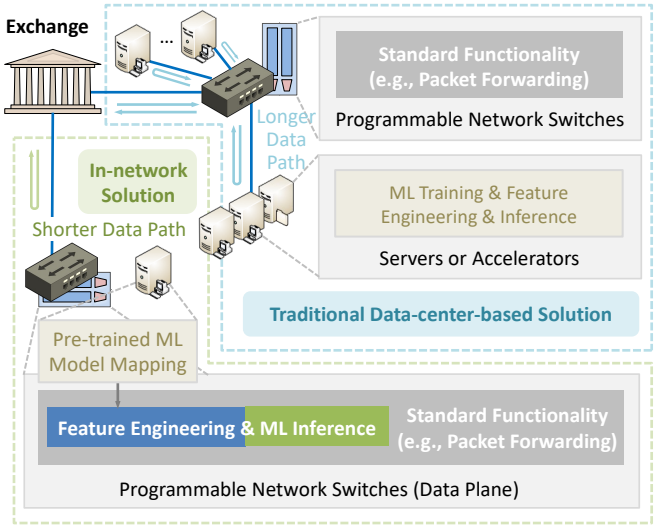


Figure 2: Comparison of in-network solutions to traditional data-center-based approaches in practical scenarios.

To summarize, the main contributions of this work are as follows:

- We introduce an innovative solution for accelerating ML-based HFT using programmable switches, offering ultra-low-latency, high-throughput, and energy-efficiency, while having a minimal effect on ML performance.
- We design and implement a prototype that performs advanced feature engineering and inference directly within the programmable network device. We integrate feature engineering and extraction workflow with ML-related processes, deploying it on both software and off-the-shelf hardware switches to demonstrate its practicality, based on high-frequency market feeds.
- We demonstrate, for the first time, the use of in-network ML for stock market volatility movement forecasting.
- We evaluate the solution for both prediction and system-level networking performances within a networked-system testbed. Using five in-network ML algorithms with different data sources to predict stock price and volatility movements, we show that our solution achieves better performance than state-of-the-art in-network ML solutions.
- We examine a hybrid deployment strategy, evaluating its impact on accuracy, error rate, the fraction of inferences managed by the switch, and latency. Our findings indicate that a hybrid deployment enhances prediction performance while maintaining ultra-low latency benefits.

2 Related Work

ML for Market Prediction. ML has shown its effectiveness in enhancing various facets of the trading process, such as market forecasting, trading signal generation, and optimization of order execution [22]. Among these, accurate predictions of short-term stock price and volatility trends are crucial for trading entities and market makers, as they lead to improved decision-making and fairer pricing of financial derivatives for end investors. Yet, market prediction is challenging due to the myriad of influencing factors [34]. Researchers have utilized a range of ML strategies [12, 35, 13, 48], leading to models that, while more accurate, also grow in complexity and data demand, presenting difficulties for traditional computing platforms and slowing down processing speeds. Given that a competitive edge in HFT hinges on achieving ultra-low latency, a key challenge lies in accelerating ML-based market predictions while maintaining their accuracy.

Hardware Acceleration for Trading. To date, various hardware devices like application-specific integrated circuits (ASICs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs) have been employed to speed up trading processes [33]. While FPGAs have been utilized in prior studies to accelerate market data processing and trading applications [31, 45], other approaches have combined customized network interface cards (NICs) and software optimizations for latency improvements [44]. The industry has also made notable advances in hardware acceleration to support financial services [38, 9]. However, hardware accelerators, while integral to server CPUs, elevate power consumption due to their energy demands. Conversely, inherently programmable network devices, already existing within the network, avoid additional power costs, facilitating in-network ML without extra hardware. Yet, research on accelerating ML-driven trading directly within the network itself is still limited. Given the superior processing speed of ASICs over other network devices [28], this study mainly adopts switching ASICs to achieve the lowest end-to-end latency.

In-network ML. Network programmability has driven network evolution, notably through software-defined networking (SDN), which allows for software-based network management [41]. The development of the Programming Protocol-independent Packet Processors (P4) language enables customizable packet processing and forwarding by network devices [16]. The data plane of network devices, responsible for the actual forwarding of network packets, is programmable through P4. Furthermore, the Protocol-Independent Switching Architecture (PISA) is a programmable match-action (M/A) pipeline architecture that enables granular control over packet processing [30]. Certain P4 targets, like the behavioral model (BMv2) switch [1] and P4 platform on Raspberry Pi (P4Pi) [26], utilize standard CPUs for executing packet forwarding programs, whereas others leverage hardware such as FPGAs [23], switches (e.g., Intel Tofino [6]), and NICs (e.g., NVIDIA BlueField [3]). P4 programs, by running on target devices, enable data plane programmability and facilitate the offloading of server applications to network devices, thus capitalizing on their lower power overheads and enhanced processing efficiency [47]. Previous research has focused on enhancing ML applications by offloading some parts of ML functions into the data plane such as feature extraction and weight aggregation [40, 27]. To date, various ML models have been adapted for in-network ML, broadening its applicability across fields [55]. Nonetheless, the scope of in-network ML applications remains narrow, particularly in finance, with a few studies showcasing its utility in financial market prediction [55, 20, 21]. These works focused on a single use case with the utilization of basic MBO and LOB data features, resulting in suboptimal ML model performance. This highlights the need for further feature engineering, which can offer more detailed insights and enhance model efficacy.

3 System Design

The PISA architecture in a programmable switch consists of three key building blocks: a parser, a deparser, and an M/A pipeline [30]. The parser functions as a state machine that extracts a sequence of fields from a packet into what is known as a packet header vector (PHV). This PHV then undergoes processing in the pipeline through a series of logical stages using M/A tables. These tables are used for looking up key values and mapping them to specific actions that

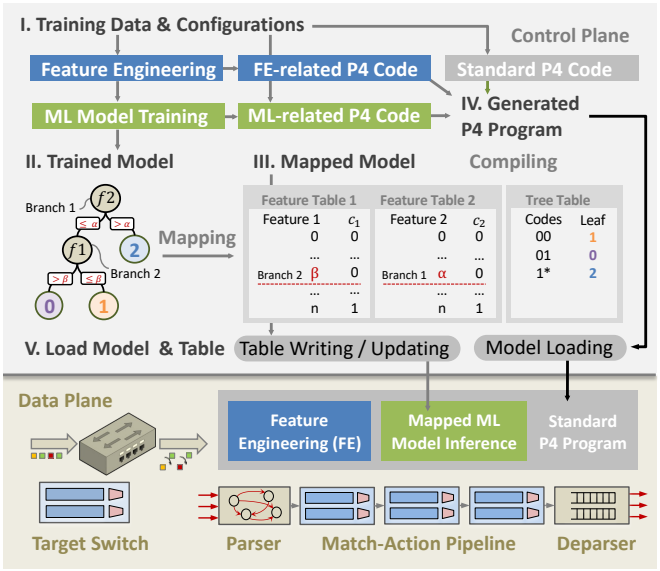


Figure 3: System design of in-network solutions.

dictate how the packet is processed. Finally, the deparser reassembles the PHV with the packet payload before the packet is sent out.

Our solution, designed for networked environments, accelerates ML-based market forecasts by conducting feature engineering and ML inference directly within switches. More specifically, this approach involves the selection, manipulation, and transformation of raw market data into richer features entirely within the data plane. These features are then used for inference, with the capability for some to be retained by leveraging registers for stateful analysis, along with fields from upcoming packets. The system architecture design of our solution is illustrated in Figure 3, including components and processes on the control plane and the data plane. The control plane governs how data is forwarded within a network, whereas the data plane executes the actual forwarding of packets.

An in-network deployment begins with offline ML training on servers, using historical market feeds along with both raw and engineered data fields as features (as depicted in Figure 3 part I). This trained model (part II) is then converted into a format compatible with the data plane using M/A table constructs (part III). A P4 program is generated from the mapping, tailored to the specific architecture of the target platform (part IV). This program is comprised of codes related to the *architecture*, including standard network switching functions, *feature engineering*, which varies by tasks and datasets, and *inference models*, created using current mapping methodologies [55]. Upon compilation, the program is deployed onto the target platform, with table entries loaded via the control plane (part V). This design enables obtaining prediction results directly within the data plane, based on pre-trained models.

Parts II & III of Figures 3 illustrate an example of how the inference process of a decision tree (DT) is mapped onto the data plane. This involves deconstructing the model into a tree-like structure of nodes and branches (part II), exemplifying a partitioning of the feature space using two features, $f1$ and $f2$. The inference process tracks a data sample's path through the tree, based on comparisons of input feature values to threshold values, leading to specific branches. To adapt this structure for the data plane, M/A table rules are established, translating the path of feature splits into feature tables and a singular tree table (part III). These tables use input features and splits for matching conditions, with actions encoded to guide data through the model's structure, leading to decisions informed by match conditions recorded as code pairs in the tree table.

Following offline model training and mapping, ultra-low-latency feature engineering and ML inference can be realized in conjunction with network forwarding. The nature of feature engineering varies with each feed due to raw data differences, requiring customized approaches for creating new features. Detailed examples and further discussions can be found in Section 4. However, the adaptability of feature engineering is subject to the limitations of programmable data planes within switches, which lack the capability for complex calculations, including trigonometric, logarithmic, and exponential functions, as well as advanced operations like matrix multiplication or calculus. Section 4 delves into how these constraints are navigated to implement feature engineering effectively within limitations.

This design also enables a hybrid deployment to further improve prediction performance, as suggested in [54]. Under this setup, predictions made within programmable switches are only labeled if they reach a high classification confidence level. If not, the market feeds are forwarded to a backend where a larger model performs the inference. This approach helps to lower latency for the majority of inferences, maintaining high prediction accuracy without sacrificing networking performances.

4 Implementation

Our solution leverages the P4 language for implementation on two different switches, BMv2 software switch [1] and Intel Tofino switch-ASIC [6]. BMv2 is an open-source behavioral software switch and serves as a flexible testing ground due to its minimal resource restrictions, making it ideal for evaluating functionalities and P4 prototyping. Intel Tofino, our second target, is a high-performance programmable switch-ASIC capable of multi-terabits per second data rates and sub-microsecond latency [10], suitable for deployment at the network edge. Similar to other programmable switch-ASICs, Tofino has limited memory and processing stages [17]. While BMv2 is a suitable choice for P4 prototyping, utilizing Tofino showcases the viability of employing off-the-shelf platforms in real-world trading scenarios. ML models are trained on servers using Python and the scikit-learn (sklearn) library [36].

4.1 Key Challenges and Solutions

Given the limitations inherent to programmable hardware, feature engineering must be adapted to fit the specific capabilities of the switches. Below, we list several main challenges that may arise when enabling in-network feature engineering processes, along with proposed solutions used for implementing the in-network approach.

Handling Non-Supported Data Types. P4 is tailored for packet processing and forwarding, focusing on the manipulation of network packet headers' fixed-size fields, typically integers. Consequently, P4-compatible programmable data planes lack built-in support for floating-point numbers, which are often used to represent values like prices in market feeds. These numbers are converted to integers in P4 data planes, using truncation or quantization.

Overcoming Limited Mathematical Operations Support. P4's support is confined to a basic set of operations, including addition, subtraction, logical conjunction (AND), disjunction (OR), exclusive disjunction (XOR), and bit shift. However, P4 does not support other common operations crucial for feature engineering such as direct variable comparisons. To circumvent these limitations, different approaches are used to perform the operations indirectly. For instance,

variable comparisons within if-else conditions are executed by calculating the difference between variables and comparing this difference to zero. Complex calculations, like multiplication and division, are facilitated through M/A table lookups. By using operands as keys for table lookups, intricate feature engineering tasks are achievable, with lookup tables aggregating multiple entries under common attributes to enhance efficiency and scalability.

Enabling Stateful Analysis. Feature engineering sometimes necessitates aggregating data from multiple network transactions rather than relying solely on data from an individual packet. For this purpose, registers can be used to conduct stateful analysis. However, a register can be accessed only once in the pipeline. This allows for a single read-modify-write operation per register. As some feature engineering processes with complex updates require reading from the register at the beginning of the pipeline and then writing back to it later in the same pipeline, a single access is insufficient. Packet recirculation, meaning that packets re-enter the pipeline for another processing round, offers a solution for this challenge, with the first pass used to read the register and the second for writing, or vice versa. This mechanism also helps tackle resource limitations when feature engineering tasks become resource-intensive, especially when dealing with unsupported constructs like loops or performing extensive price and quantity analysis from LOBs. However, recirculation comes at the cost of higher latency.

4.2 Data Plane Implementation

Our P4 solution incorporates feature engineering code with ML inference code generated by Planter, a framework for model mapping to programmable network devices [55]. Feature engineering is performed based on raw LOB data, developing new features such as the mid-price, bid-ask spread, and order flow imbalance. Each can reveal the direction and momentum of market sentiment from unique perspectives, making them efficient features for market forecasting.

A bid order is an offer to buy a specific quantity of a financial instrument at or below a specified price, while an ask order is to sell at a certain price or higher, with both kinds of orders placed on the respective sides of a LOB [50]. As the raw form of LOB data, the LOB state at time t can be defined as the vector of price and volume information for the top n bid and ask price levels:

$$\mathbf{s}_t^n := \left(p_t^{1,a}, q_t^{1,a}, p_t^{1,b}, q_t^{1,b}, \ldots, p_t^{n,a}, q_t^{n,a}, p_t^{n,b}, q_t^{n,b}\right)^\top \in \mathbb{R}^{4n} \tag{1}$$

where $p_t^{i,a}$ and $p_t^{i,b}$ are the ask and bid prices at the i^{th} level at time t while $q_t^{i,a}$ and $q_t^{i,b}$ are the respective quantities.

The mid-price is the average of the highest bid price and the lowest ask price [32], whereas the bid-ask spread is known as the difference between the highest bid price and lowest ask price [43]. These two features at time t can be computed as follows:

$$P_t := \frac{p_t^{1,a} + p_t^{1,b}}{2} \tag{2}$$

$$S_t := p_t^{1,a} - p_t^{1,b} \tag{3}$$

The order flow imbalance quantifies the disparity between bid and ask order volumes within a specific time frame. Given two consecutive LOB states for the top n price levels at time $t-1$ and t of the form (1), the ask and bid order flows at time t can be defined as the vectors $\mathbf{A}_t, \mathbf{B}_t \in \mathbb{R}^n$ [25], where each element is given by:

$$\mathrm{A}_{t,i} := \begin{cases} -q_{t-1}^{i,a}, & \text{if } p_t^{i,a} > p_{t-1}^{i,a}, \\ q_t^{i,a} - q_{t-1}^{i,a}, & \text{if } p_t^{i,a} = p_{t-1}^{i,a}, \\ q_t^{i,a}, & \text{if } p_t^{i,a} < p_{t-1}^{i,a}, \end{cases} \tag{4}$$

$$\mathrm{B}_{t,i} := \begin{cases} q_t^{i,b}, & \text{if } p_t^{i,b} > p_{t-1}^{i,b}, \\ q_t^{i,b} - q_{t-1}^{i,b}, & \text{if } p_t^{i,b} = p_{t-1}^{i,b}, \\ -q_{t-1}^{i,b}, & \text{if } p_t^{i,b} < p_{t-1}^{i,b}, \end{cases} \tag{5}$$

The order flow imbalance can be determined by subtracting $\mathbf{B}_t$ and $\mathbf{A}_t$ as follows:

$$\mathbf{I}_t := \mathbf{B}_t - \mathbf{A}_t \in \mathbb{R}^n \tag{6}$$

Formulas (4), (5), and (6), represent nonlinear transformations that are widely used (on CPUs/GPUs) to convert the non-stationary time series of LOB states into stationary ones [18, 25].

Compared to calculating the mid-price and bid-ask spread, obtaining order flow imbalance on a programmable data plane is complex. The pseudocode for deriving order flow imbalance on the data plane is outlined in Algorithm 1. A register is used to record the price and quantity details on both sides of the most recent order. These variables are stateful, requiring ongoing maintenance and updates. Other variables in Algorithm 1 are user-defined metadata, used as intermediates for indirect comparisons in feature engineering. The output includes two features: one that captures the value of order flow imbalance and another that indicates whether it is positive or negative. These features are assigned based on the outcomes of comparing prices and quantities between sides and time.

Algorithm 1 Compute order flow imbalance at i_{th} price level

- $R_{t-1}[i]$: A register array storing price and quantity information on both sides at the i^{th} price level at time $t-1$.
- $p_t^{i,a}, p_t^{i,b}$: Ask and bid prices at the i^{th} level at time t.
- $q_t^{i,a}, q_t^{i,b}$: Ask and bid quantities at the i^{th} level at time t.
- $c, d, e, f, g, h, j, k, l, m, n$: Temporary variables for performing indirect comparisons.
- f_1, f_2: Order flow imbalance related features.

function COMPUTATION($p_t^{i,a}, q_t^{i,a}, p_t^{i,b}, q_t^{i,b}$)
    $p_{t-1}^{i,a}, q_{t-1}^{i,a}, p_{t-1}^{i,b}, q_{t-1}^{i,b} \leftarrow R_{t-1}[i]$ ▷ Read price and quantity information on both sides at the I^{th} level at time $t-1$.
    $R_{t-1}[I] \leftarrow p_t^{i,a}, q_t^{i,a}, p_t^{i,b}, q_t^{i,b}$ ▷ Write back updated price and quantity information on both sides at the I^{th} level at time t.
    $c, d, e, f, g, h, j, k, l, m, n \leftarrow p_t^{i,a} - p_{t-1}^{i,a}, p_t^{i,b} - p_{t-1}^{i,b}, q_t^{i,b} - q_t^{i,a}, q_{t-1}^{i,a} - q_{t-1}^{i,b}, q_t^{i,a} + q_{t-1}^{i,b}, q_{t-1}^{i,a} + q_t^{i,b}, q_t^{i,b} - g, q_{t-1}^{i,a} - g, h - q_t^{i,a}, h - q_{t-1}^{i,b}, h - g$ ▷ Compare prices and quantities.
    if $c < 0$ **and** $d < 0$ **then** ▷ If $p_t^{i,a} < p_{t-1}^{i,a}$ and $p_t^{i,b} < p_{t-1}^{i,b}$.
        $f_1, f_2 \leftarrow q_t^{i,a} + q_{t-1}^{i,b}, 0$ ▷ Assign values to f_1 and f_2.
        … **…** ▷ Four cases are omitted here for brevity.
    else if $c == 0$ **and** $d < 0$ **and** $k < 0$ **then** ▷ Else if $p_t^{i,a} == p_{t-1}^{i,a}$ and $p_t^{i,b} < p_{t-1}^{i,b}$ and $q_{t-1}^{i,a} < q_t^{i,a} + q_{t-1}^{i,b}$.
        $f_1, f_2 \leftarrow g - q_{t-1}^{i,a}, 0$ ▷ Assign values to f_1 and f_2.
        … **…** ▷ Nine cases are omitted here for brevity.
    else if $c > 0$ **and** $d > 0$ **then** ▷ Else if $p_t^{i,a} > p_{t-1}^{i,a}$ and $p_t^{i,b} > p_{t-1}^{i,b}$.
        $f_1, f_2 \leftarrow h, 1$ ▷ Assign values to f_1 and f_2.
    return $f1, f2$ ▷ Return order flow imbalance as features.

The features above are used solely for illustrative examples. Additional feature engineering and the development of other features based on market feeds can be further performed within data planes, tailored to meet the requirements of data formats and task objectives.

5 Experiments

In this section, we conduct experiments on different financial market data feeds and compare our in-network solution with server-based benchmarks, examining their ML capabilities, latency, and throughput. Additionally, performance comparisons are made among the five most commonly used in-network ML algorithms. The results indicate that the in-network solution is capable of accelerating ML while maintaining forecasting accuracy across datasets and programmable network targets.

Datasets and Tools: Evaluation is conducted on two financial market feed datasets: NASDAQ's Historical TotalView-ITCH sample data feeds [7] and LOBSTER dataset [8], focusing specifically on the stock AAPL (Apple Inc). Owing to the lack of feeds spanning consecutive days in the NASDAQ's sample data source, evaluation is based on information from a single day, specifically the most recent data available on January 30, 2020, at the time of analysis. The LOBSTER dataset contains raw LOB data over consecutive periods, with the latest continuous four-day stretch, from December 27, 2022, to December 30, 2022, being used. Our analysis emphasizes comparing the relative performance of in-network models to server benchmarks instead of assessing absolute performance, making a set of around 13.44 million total entries sufficient for our purposes without affecting the findings. An open-source tool [14] is used for reconstructing LOBs from MBO data feed messages, and a framework [55] is used for in-network ML deployment.

Experiment Setup: An APS-Networks BF6064T-X Intel Tofino switch equipped with 64×100G ports and SDE 9.4.0 software is used in our experiments. Server-based evaluation, including network traffic generation, utilizes two ASUS ESC4000A-E10 servers, both powered by AMD EPYC 7302P CPUs and equipped with 256GB of DDR4 RAM, running on Ubuntu 20.04 LTS. They are connected to the switch through NVIDIA ConnectX-5 100G NICs and direct attach cables. For server-based ML training and inference, the sklearn library [36] is applied.

5.1 Prediction Performance

The predictive accuracy is evaluated using five ML algorithms that are commonly applied to market forecasting tasks, including k-means (KM), k-nearest neighbors (KNN), decision trees (DTs), random forests (RFs), extreme gradient boosting (XGB) [12, 35, 13, 48]. Model training uses up to the top ten price levels of LOBs in input feeds for server-based predictions. For in-network predictions on switches, only the best price level of LOBs is used for feature engineering, due to resource constraints on the programmable hardware. Server-based benchmarks are trained with unlimited-size models while in-network models are of limited size. The labels are used to predict future stock mid-price or volatility movement (up, down, and stationary) over the next 100 ticks. Mid-price is computed following Formula (2) while the rolling standard deviation of log returns is calculated as a measure of volatility. A smoothing labeling approach [32] is employed to extract more consistent signals from highly stochastic feeds and oversampling techniques are applied to address class imbalance.

Table 1 presents the prediction performance results leveraging the most common ML evaluation metrics, including macro precision (PRE), macro recall (REC), macro F1-score (F1), and accuracy (ACC). Despite some performance differences between in-network solutions and server-based benchmarks across different models, it is notable that in 85% of the instances, the performance loss for F1-scores is less than 7% on a programmable switch. In certain cases, in-network models demonstrate performance that is on par with their server-based counterparts, exhibiting minimal discrepancy in results. Where there is a greater gap, it is generally due to the losses incurred when mapping models to fit within switches, the constraints on model size, and a restricted number of engineered features, all because of limitations on hardware capacity.

Table 1: Prediction performance (%) with different datasets and use cases.

Models	In-network Solutions				On-server Benchmarks			
	PRE‡	REC‡	F1‡	ACC‡	PRE‡	REC‡	F1‡	ACC‡
Mid-price Movement Prediction with NASDAQ TotalView-ITCH								
KM	38.51	40.10	35.20	**40.60**	42.29	41.11	35.48	**39.95**
KNN	30.95	30.90	30.85	31.20	42.08	39.91	40.10	39.86
DT	**44.11**	**43.51**	**43.45**	43.85	**44.15**	**43.53**	**43.48**	43.88
RF	44.59	42.93	41.60	42.45	48.14	44.49	44.13	44.64
XGB	43.43	43.32	43.32	43.31	44.64	44.60	44.60	44.58
Volatility Movement Prediction with NASDAQ TotalView-ITCH								
KM	**37.62**	**37.54**	31.80	36.42	**36.72**	**37.44**	33.09	38.84
KNN	32.91	32.71	25.47	33.14	39.56	39.03	38.71	38.68
DT	43.68	42.81	**42.67**	**42.90**	43.39	43.29	**42.91**	**43.91**
RF	42.00	41.88	41.87	42.19	45.98	44.08	43.14	44.80
XGB	44.29	43.05	42.86	43.19	43.94	43.83	43.57	44.37
Mid-price Movement Prediction with LOBSTER								
KM	22.92	33.45	22.54	33.46	25.59	37.75	29.43	37.67
KNN	32.17	**33.04**	29.67	**33.09**	34.75	**34.74**	34.53	**34.73**
DT	43.99	43.24	43.35	43.25	46.40	45.36	45.47	45.36
RF	42.93	42.99	42.56	42.97	45.93	45.90	45.90	45.89
XGB	**44.62**	44.11	**44.23**	44.11	**46.62**	46.21	**46.32**	46.21
Volatility Movement Prediction with LOBSTER								
KM	**23.35**	**33.45**	**24.37**	38.54	**23.74**	**33.71**	**26.22**	34.16
KNN	13.17	33.33	18.88	**39.51**	34.08	33.88	32.33	**32.60**
DT	42.27	40.00	36.60	43.39	45.69	44.60	42.63	47.27
RF	39.70	38.56	38.15	40.13	45.78	44.63	42.90	47.67
XGB	40.17	39.16	37.87	41.86	45.46	45.62	43.89	47.85

‡ **Bold** and underline signify the highest and second-highest performance of each metric, respectively, when comparing in-network models to those based on servers. This corresponds to the least and second-least performance loss.

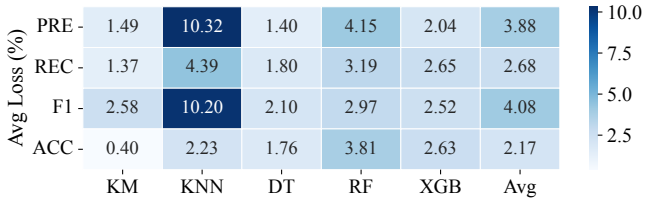


Figure 4: The average (avg) performance loss of different models and metrics across datasets and use cases, in-network solutions relative to server-based benchmarks.

The heatmap depicted in Figure 4 shows the average performance loss of different models and metrics across datasets and use cases. It reveals that, except for KNN, all in-network models demonstrate robust performance across these scenarios, maintaining an average performance decrease within 5% for all evaluated metrics. Specifically, the average losses in precision, recall, F1-score, and accuracy are 3.88%, 2.68%, 4.08%, and 2.17%, respectively. In-network KNN's performance is less optimal due to its sensitivity to model depth [53]; a shallow depth fails to effectively simulate the classification boundary, whereas a deep model cannot fit within the data plane.

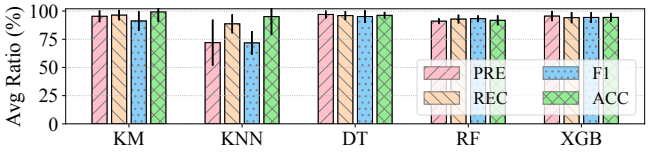


Figure 5: The average (avg) performance ratio of different models and metrics across datasets and use cases, in-network solutions relative to server-based benchmarks.

Upon closer analysis of the ML efficiency of our in-network approaches, the average performance ratios of different models and metrics across datasets and use cases are computed, as shown in Figure 5. Specifically, in-network solutions attain, on average, 90.18% precision, 93.62% recall, 89.14% F1-score, and 95.34% accuracy relative to server benchmarks. These results, consistent with those presented in Figure 4, demonstrate that our in-network approaches closely match server-based models in terms of performance, with minimal loss and consistent stability across different scenarios.

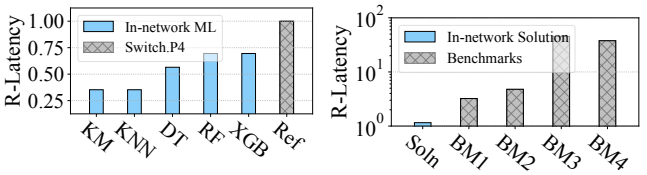
Additionally, to provide a performance comparison between existing in-network ML algorithms and deep learning models suitable for sequence prediction tasks, a server-based recurrent neural network (RNN) model using the long short-term memory (LSTM) architecture is applied to the same datasets. This model, renowned for capturing long-term dependencies, is constructed using TensorFlow and Keras, comprising an LSTM layer of 50 units, followed by a dropout layer to prevent overfitting and a dense output layer with softmax activation for multi-class classification. The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss. While direct implementation into switches remains impractical so far due to the computational demands of LSTM or other types of RNN, the majority of evaluated in-network models exhibit comparable efficacy to it. The latter, for example, yields an F1-score of 43.06% and an accuracy of 44.67% when predicting stock mid-price movement using NASDAQ TotalView-ITCH data feeds, while using LOBSTER feeds results in an F1-score of 45.91% and an accuracy of 46.25%.

5.2 *Networking Performance*

To assess system-level networking performance, an evaluation is performed directly on the data plane of Intel Tofino [6], focusing on two key indicators: latency and throughput. Latency measures the time it takes for a data packet to travel from its origin to the endpoint via a hardware network device, while throughput refers to the amount of data that can be transmitted over the device within a given period.

Due to Tofino-related non-disclosure agreements, the relative pipeline latency (R-Latency) of our in-network solution is compared with that of the Intel reference switch program, namely *switch.p4*, both determined by Tofino's compiler. The *switch.p4* program details how a network switch operates, covering basic functions from simple network switching to more complex routing tasks [5]. Figure 6 (a) illustrates that all evaluated in-network models achieve a latency that is lower than 75% of that observed with the reference *switch.p4*, demonstrating that even under resource constraints, the in-network solution still achieves comparable latency to simple packet switching. Furthermore, it is verified that this solution can be deployed alongside other networking functions.

Additionally, to measure the framework latency of our solution, experiments are conducted using two servers connected through a programmable switch. The framework R-Latency is calculated by comparing the latency observed with the deployment of our solution to that of simple packet forwarding via the switch. These measurements utilize the Precision Time Protocol (PTP) alongside the ptp4l toolkit. The results, illustrated in Figure 6 (b), show that our solution achieves microsecond-level average latency across models, yielding an improvement of 64% to 97% compared to state-of-the-art benchmarks, which include NIC-based, FPGA-based, and server-based data processing. Specifically, our solution has a lower latency than the end-to-end latency observed in a NIC-based trading system [44], the turnaround latency in FPGA-based and CPU-based NASDAQ data feed handlers [37], and the minimum latency attained by the fastest NASDAQ traders [15]. Considering that even a slight rise in latency for trading applications can result in the loss of millions of dollars from missed arbitrage opportunities [29], this finding suggests that in-network ML holds significant potential for HFT.



(a) Pipeline R-Latency (b) Framework R-Latency

Figure 6: (a) The relative latency (R-Latency) of programmable switches' pipeline for different in-network models, measured for standalone ML and standalone $switch.p4$. (b) The R-Latency of our solution (Soln) and four benchmarks (BM1 [44], BM2 [37], BM3 [37], BM4 [15]), relative to simple packet forwarding through a switch.

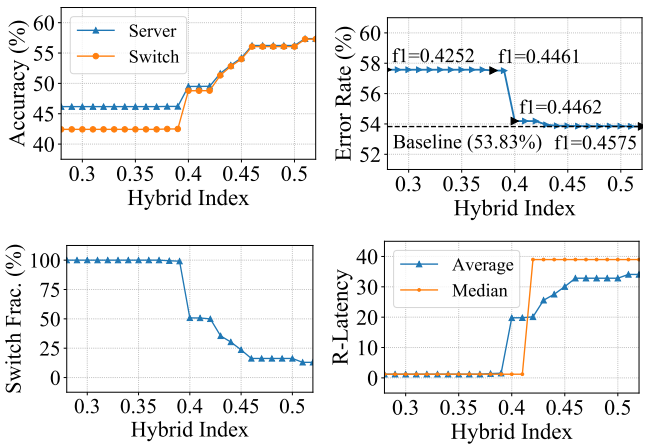
In a throughput evaluation, a snake configuration is employed in the setup, allowing traffic to flow in a continuous loop from one port to the next, thus facilitating connectivity through all 64 ports. This is achieved using Pktgen version 21.03.0 [4] powered by DPDK version 20.11.1 [2] for packet generation. The findings demonstrate that our in-network solution is capable of reaching the line rate on a commercial Tofino switch, achieving a throughput of 6.4Tbps (Terabits per second) across all tested cases.

The profitability of our in-network solutions compared with server-based benchmarks also considers both gains and losses. A simulation indicates that while our solution may incur a 2.17% annual profitability loss due to ML performance degradation, it can generate an additional 4.08% profit from latency reduction. Overall, this results in an average additional profit of 1.91% per year, with a profit/loss ratio of 1.88. In the $7-$8 billion HFT market, this translates to over $100 million additional profits.

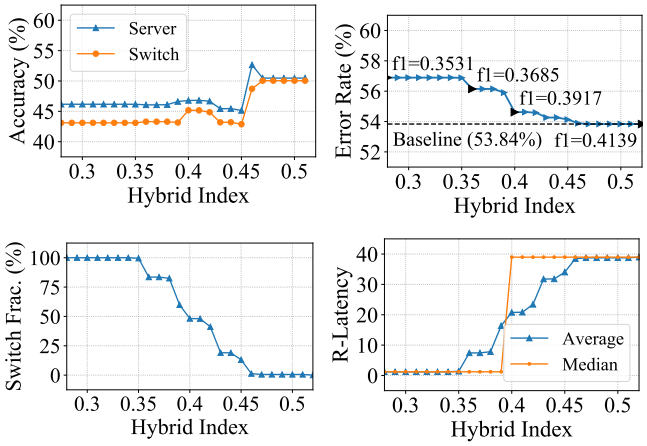
5.3 *Hybrid Deployment Performance*

The predictive performance of our in-network solution can be further enhanced through a hybrid deployment approach [54]. The baseline is a full-sized ensemble model running on servers. Meanwhile, a smaller model is deployed on the switch for inference. Messages classified with low confidence by the switch are then sent to the servers for more in-depth classification. A hybrid index is defined as a particular confidence threshold set on the switch, determining which classifications are processed on the switch and which are escalated to the servers.

Figure 7 depicts the correlation between the hybrid index and four critical performance indicators: accuracy, misclassification rate, the proportion of offloaded traffic, and relative framework latency relative to simple forwarding through the switch. This is demonstrated using random forest models for classification, handling two predictive tasks on LOBSTER data feeds.



(a) Mid-price Movement Prediction



(b) Volatility Movement Prediction

Figure 7: The accuracy, error rate, and fraction (Frac.) of traffic offloaded by the switch, and end-to-end average and median relative latency (R-latency) of the in-network solution with LOBSTER using random forest models, across hybrid indices.

Several common findings can be identified from Figure 7: Firstly, as the hybrid index is increased, there is a noticeable improvement in prediction accuracy both on the switch and the server. With a higher index, the misclassification rate decreases and the F1-score improves. However, the classification accuracy on the switch is lower than that on the server because the switch's model, which contains only 4 trees with a maximum depth of 3 and a maximum of 1000 leaf nodes, is substantially smaller than the server's model. The server's model comprises 200 trees with a maximum depth of 30 and up to 10000 leaf nodes. Secondly, setting a higher hybrid index means more order feeds are directed to the server for processing, reducing the fraction of traffic that the switch can offload. Thirdly, this redirection of more orders to the server also results in higher average latency per processed order. The significant increase in median latency noted in related figures highlights the point where half of the orders are routed to the server, with the other half managed solely on the switch.

As Figure 7 (a) demonstrates, for mid-price movement prediction, the optimal hybrid index is 0.42, as this setting allows 49.98% of the traffic to be managed directly on the switch, with the error rate increase capped at 0.3% compared to the server-based baseline. In such scenarios, the average latency per message is reduced by 39.91% relative to a server-only setup. According to Figure 7 (b), for predicting volatility movements, the ideal hybrid index of 0.41 is recommended. Adjusting to this level keeps the error rate increase below 1.3%, allows 48.05% of the traffic to be processed on-switch, and achieves a latency reduction of 49.17% on average. In practice, an appropriate hybrid index can be selected case by case to balance all metrics.

Overall, our evaluation indicates that a hybrid approach processes approximately 49% of traffic directly on the switch, avoiding the need for server intervention. This strategy not only keeps the average change in the misclassification rate under 0.8% but also achieves an average latency reduction of 45%, proving its effectiveness in handling large-scale trading feeds.

6 Discussion

In-network ML Algorithms: This work focuses on five commonly-used in-network ML models. It excludes others, like in-network support vector machines and naive Bayes, due to higher hardware resource consumption demands, which would hinder the simultaneous deployment of feature engineering code (although direct inference with these models is feasible). Despite the growing popularity of deep learning and reinforcement learning in trading, they are not yet sufficiently developed for in-network applications [53]. However, the prospect of integrating a wider range of algorithms into data planes is promising in future research, which could further enhance in-network ML's capabilities and applicability in more use cases.

Low-latency Network Devices: By design, this work focuses on accelerating ML for short-term financial market predictions using raw LOB data directly within the network. Our solution stands out by eliminating the latency involved in reaching the end host, with negligible overhead added compared to standard network forwarding processes. The adoption of low-latency programmable switches could further reduce latency. The promising capacity of in-network ML to lower latency opens avenues for developing new trading system architectures that incorporate programmable network devices.

7 Conclusion

This study presented a novel approach to accelerate ML for trading using programmable network switches. Based on raw LOB feeds, an in-network solution was designed to perform feature engineering and inference directly within the programmable data plane. The findings demonstrate that our solution significantly reduces end-to-end latency to microsecond scales by 64% to 97% and achieves a throughput of 6.4Tbps while minimizing average loss in all ML metrics to between 2% and 4%, compared to baselines. Moreover, employing a hybrid deployment strategy enhances its predictive capabilities. Our experiment shows that a hybrid setup enables on-switch processing of 49% of the traffic and an average reduction of 45% in latency while limiting the increase in misclassification rate to below 0.8%. In summary, this methodology streamlines integrating new data feeds and algorithms, possessing the potential for straightforward extension and application across various ML-based, time-sensitive financial use cases in the future.

Acknowledgements

This work was partly funded by VMware. We acknowledge support from Intel and NVIDIA.

References

[1] Reference P4 software switch. https://github.com/p4lang/behavioral-model.

[2] DPDK. https://www.dpdk.org/.

[3] NVIDIA BlueField networking platform. https://www.nvidia.com/en-gb/networking/products/data-processing-unit/.

[4] Pktgen. https://pktgen-dpdk.readthedocs.io/en/latest/contents.html.

[5] switch.p4. https://github.com/p4lang/switch/tree/master/p4src.

[6] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[7] NASDAQ ITCH data source. 2020. URL https://emi.nasdaq.com/ITCH/Nasdaq%20ITCH/.

[8] LOBSTER data source. 2022. URL https://lobsterdata.com/.

[9] AMD. AMD announces new Alveo X3 series for electronic trading. *HPC Wire*, 2022. URL https://www.hpcwire.com/off-the-wire/amd-announces-new-alveo-x3-series-for-electronic-trading/.

[10] APS Networks. BF6064X-T advanced programmable switch. https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF6064X-T_V04.pdf[Online, accessed Feb-2023].

[11] J. Arifovic et al. Machine learning and speed in high-frequency trading. *Journal of Economic Dynamics and Control*, 139:104438, 2022.

[12] M. Ballings, D. Van den Poel, N. Hespeels, and R. Gryp. Evaluating multiple classifiers for stock price direction prediction. *Expert systems with Applications*, 42(20):7046–7056, 2015.

[13] S. Basak, S. Kar, S. Saha, L. Khaidem, and S. R. Dey. Predicting the direction of stock market prices using tree-based classifiers. *North Am. J. Econ. Finance*, 47:552–567, 2019.

[14] M. Bernasconi-De-Luca et al. martinobdl/ITCH: ITCH50Converter, 2021. URL https://zenodo.org/record/5209267.

[15] J. Bonart and M. D. Gould. Latency and liquidity provision in a limit order book. *Quant. Finance*, 17(10):1601–1616, 2017.

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[17] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 15–29, 2019.

[18] R. Cont, A. Kukanov, and S. Stoikov. The price impact of order book events. *Journal of financial econometrics*, 12(1):47–88, 2014.

[19] E. A. Gerlein, M. McGinnity, A. Belatreche, and S. Coleman. Evaluating machine learning classification for financial trading: An empirical approach. *Expert Systems with Applications*, 54:193–207, 2016.

[20] X. Hong, C. Zheng, S. Zohren, and N. Zilberman. Linnet: limit order books within switches. In *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, pages 37–39. 2022.

[21] X. Hong, C. Zheng, S. Zohren, and N. Zilberman. LOBIN: In-network machine learning for limit order books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, pages 159–166. IEEE, 2023.

[22] B. Huang, Y. Huan, L. D. Xu, L. Zheng, et al. Automated trading systems statistical and machine learning methods and hardware implementation: a survey. *Enterprise Information Systems*, 13(1):132–144, 2019.

[23] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4→NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the ACM FPGA 2019*, pages 1–9, 2019.

[24] M. Kearns and Y. Nevmyvaka. Machine learning for market microstructure and high frequency trading. *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.

[25] P. N. Kolm, J. Turiel, and N. Westray. Deep order flow imbalance: Extracting alpha at multiple horizons from the limit order book. *Mathematical Finance*, 33(4):1044–1081, 2023.

[26] S. Laki, R. Stoyanov, D. Kis, R. Soulé, P. Vörös, and N. Zilberman. P4Pi: P4 on Raspberry Pi for networking education. *ACM SIGCOMM Computer Communication Review*, 51(3):17–21, 2021.

[27] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[28] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *2012 IEEE 20th annual symposium on high-performance interconnects*, pages 9–16. IEEE, 2012.

[29] R. Martin. Wall street's quest to process data at the speed of light. *Information Week*, 4(21):07, 2007.

[30] N. McKeown. *PISA: Protocol Independent Switch Architecture*, 2015. P4 Workshop.

[31] G. W. Morris, D. B. Thomas, and W. Luk. FPGA accelerated low-latency market data feed processing. In *2009 17th IEEE Symposium on High Performance Interconnects*, pages 83–89. IEEE, 2009.

[32] A. Ntakaris, M. Magris, J. Kanniainen, M. Gabbouj, et al. Benchmark dataset for mid-price forecasting of limit order book data with machine learning methods. *Journal of Forecasting*, 37(8):852–866, 2018.

[33] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, et al. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[34] M. Obthong, N. Tantisantiwong, et al. A survey on machine learning for stock price prediction: Algorithms and techniques. 2020.

[35] J. Patel, S. Shah, et al. Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. *Expert systems with applications*, 42(1):259–268, 2015.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[37] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch, and V. Natoli. Low-latency FPGA based financial data feed handler. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96. IEEE, 2011.

[38] V. Santosh and S. Singh. Announcing DPU-based acceleration for NSX. *VMWare*, 2022. URL https://blogs.vmware.com/networkvirtualization/2022/08/announcing-dpu-based-acceleration-for-nsx.html/.

[39] D. Sanvito, G. Siracusano, and R. Bifulco. Can the network be the AI accelerator? In *NetCompute*, pages 20–25, 2018.

[40] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, et al. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[41] S. Scott-Hayward et al. SDN security: A survey. In *2013 IEEE SDN For Future Networks and Services (SDN4FNS)*, pages 1–7. IEEE, 2013.

[42] J. Stokes. One Day, the Stock Market Could Eat the Power Grid. *Wired*, 2011. URL https://www.wired.com/insights/2011/12/stock-market-power/.

[43] H. R. Stoll. Inferring the components of the bid-ask spread: Theory and empirical tests. *the Journal of Finance*, 44(1):115–134, 1989.

[44] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto. Streaming, low-latency communication in on-line trading systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[45] Q. Tang, M. Su, L. Jiang, J. Yang, et al. A scalable architecture for low-latency market-data processing on FPGA. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 597–603. IEEE, 2016.

[46] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[47] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[48] S. D. Vrontos, J. Galakis, and I. D. Vrontos. Implied volatility directional forecasting: a machine learning approach. *Quantitative Finance*, 21(10):1687–1706, 2021.

[49] Z. Xiong and N. Zilberman. Do switches dream of machine learning? toward in-network classification. In *HotNets*, pages 25–33, 2019.

[50] Z. Zhang, S. Zohren, and S. Roberts. DeepLOB: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 2019.

[51] Z. Zhang, B. Lim, and S. Zohren. Deep learning for market by order data. *Applied Mathematical Finance*, 28(1):79–95, 2021.

[52] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824*, 2022.

[53] C. Zheng, X. Hong, D. Ding, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials*, 2023.

[54] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Transactions on Networking*, 2024.

[55] C. Zheng, M. Zang, X. Hong, L. Perreault, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *ACM SIGCOMM Computer Communication Review*, 2024.