Enabling Inference Inside Software Switches

Yung-Sheng Lu and Kate Ching-Ju Lin
Department of Computer Science, National Chiao Tung University, Taiwan
{yungshenglu, katelin}@cs.nctu.edu.tw

***Abstract*—Software Defined Networking (SDN) has been emerged to solve the problem of traditional network architectures. The ability of programmable switches renders us an opportunity to have computational tasks done in the switches. With this nice property, in this work, we investigate the potential of enabling machine learning inside a network. To this end, we propose a new architecture, *Intra-Network Inference (INI)*, which equips each switch with a recently released component, called *neural compute stick* (NCS), to enable intra-switch neural network inference. Unlike conventional SDN architectures, which relay backend servers to enable inference, our INI performs inference locally at switches and, thereby, reduces the data forwarding overhead and inference latency.**

***Index Terms*—SDN, P4, Neural Networks**

I. Introduction

Software Defined Networking (SDN) has been emerged to solve the problem of traditional network architectures. In SDN, a network is divided into the control plane and the data plane. All network management is handled by the control plane by software, while switches are only in charge of simple data forwarding. Recent research has also explored how to virtualize traditional network services by Network Function Virtualization (NFV). NFV eliminates the need of providing network functions by specialized dedicated hardware (such as firewalls, routers, etc.). Instead, NFV is implemented using flexible software, which can be flexibly deployed (removed) into (out of) the network as needed. It also becomes possible to develop a variety of value-added services. By integrating NFV and SDN, more diverse services can be designed and boosted.

On the other hand, with the evolution of artificial intelligence. The system can leverage data processing or model prediction to quickly process a large amount of data and make accurate decisions. In recent years, artificial intelligence has been widely used in different fields, such as smart factories, Internet of Things, computer vision, unmanned stores and other services. Many studies [1]–[3] have also used artificial intelligence technologies to solve network management problems. If these artificial intelligence services can be implemented as a virtual network function (VNF) through software, we can make an SDN intelligent and be managed more efficiently.

However, the combination of NFV and SDN still needs to transfer data streams from switches to different virtual machines for inference, as illustrated in Fig. 1. The data exchange between switches and virtual machines would incur a fairly long delay and a large amount of data forwarding load, which would easily saturate the bottleneck link and lead to congestion. To resolve this problem, in this work, we propose

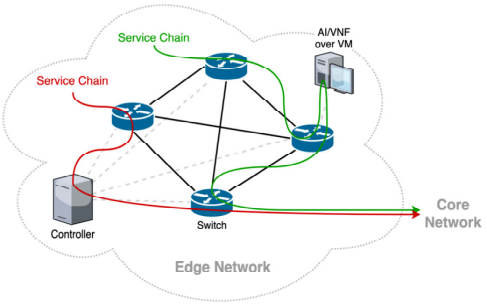

Fig. 1: Legacy Architecture for Inference

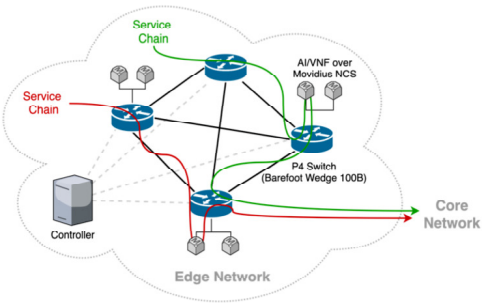

Fig. 2: Intra-Network Inference

to enable inference capability *inside switches*, as illustrated in Fig. 2. As the advanced software switches, e.g., P4, are designed to have certain computational capability, it becomes now possible to perform simple inference directly in switches, without the assistance of backend servers or virtual machines.

To achieve this goal, this paper presents a new architecture, called *Intra-Network Inference (INI)*. We develop a data forwarding processing system that allows packets to be cloned to the kernel of a switch for on-line inference. To enable in-switch inference, we leverage a new hardware, called neural compute stick (NCS), developed by Intel Movidius and released on the market recently. By connecting an NCS to a P4 switch over a USB interface, the NCS can process the cloned packets and perform real-time inference. An INI-enabled switch is capable of filtering the packets that are useful for inference and hence reduces the cost of data cloning. To verify the practicality of our design, we implement a prototype of INI using P4 switches and empirically measure the execution time required by each phase of INI.

The rest of this paper is organized as follows. Section II summarizes recent works on network management via machine learning. We then describe the design of our INI in Section III and show some preliminary results in Section IV.

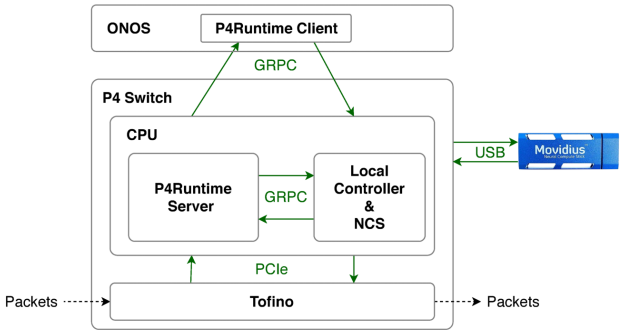

Fig. 3: INI Architecture

Finally, Section V concludes this work and summarizes some directions of future study.

II. Related Work

Recent work has shown possibility of leveraging machine learning to improve the efficiency of network management. The studies [4]–[7] propose to detect and monitor elephant flows of a network via traffic classification or rule-based algorithms. The work [4] combines switch-side filtering and controller-assisted classification to enable real-time classification. A cost-sensitive learning method [5] is then proposed to further improve the inference speed. FLight [6] alternatively develops a rule-based detector based on the the TCP communication behaviors. Later work [7] not only detects elephant flows but also counts the size of those flows for traffic engineering. The above approaches are designed mainly based on conventional machine learning classification, whose performance is sensitive to feature selection.

Some work [8], [9] investigates the network traffic classification problem. The problem of identifying end-user applications, like Facebook, Twitter and Skype is explored in [8]. [9] gives a throughout survey about the challenges of network application identification, including port abuse, random port usage and tunneling.

Recent efforts then exploit deep learning techniques to enable more network services. The work [1] enables traffic optimization, like flow scheduling, using reinforcement learning. It develops a system that consists of two components: peripheral systems (PS), which runs on all end-hosts to collect flow information and make local decision, and central system (CA), where global traffic information is aggregated and processed. The interaction between the two components mimics the design of reinforcement learning so as to optimize network performance. DDNN [10] enables distributed deep learning by allowing edges and end users to cooperatively finish the inference tasks. To reduce the number of rules in TCAM, [2] proposes a reinforcement learning scheme to determine which rules are crucial and should be kept in TCAM. A malware classification system based on CNN is developed in [3]. In this work, we select [3] as an application to demonstrate the effectiveness of our in-network inference capability.

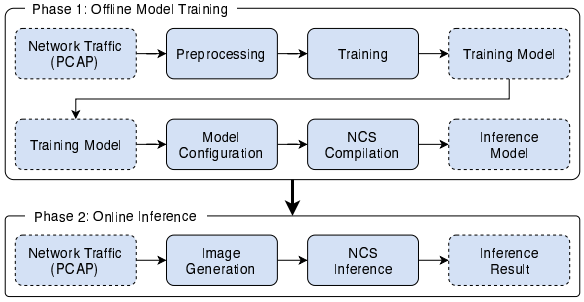

Fig. 4: Inference Process

III. INI Design

A. Architecture

Figure 3 illustrates the hardware architecture of the proposed INI. We use the Edgecore Wedge 100-32X switch with programmable Tofino switch silicon from Barefoot Networks as P4 switches. Each P4 switch is loaded with Open Network Install Environment (ONIE) software installer, which is compatible with Open Network Linux (ONL). Besides, each P4 switch also supports USB Type-A port, which can connect an NCS device. The P4Runtime Server and local controller are two main components running on a P4 switch, which communicate with each other via gRPC. gRPC is a modern open source high performance RPC framework that can run in any environment. ONOS (Open Network Operating System) is one of the controller that supports P4Runtime, which is designed for high availability, performance and scale-out. We use ONOS as the main controller that manages the whole network and let the ONOS and P4 switch communicate via gRPC.

Figure 4 illustrates the framework of INI. Our framework consists of two phases. The first phase is offline model training, which creates the inference model for the next phase. We use a CNN architecture LeNet-5 for offline training since the NCS now can only support the CNN model. Before training, we need to process the traffic and partition packets into sessions, each of which is defined as a bidirectional flow, including both directions of traffic. Since the size of input data for a CNN model should be fixed, we extract only the first n bytes (e.g., $n = 784$) from each session for model training and inference. In general, the first few bytes of a session usually includes the important connection information (e.g., MAC layer and network layer headers) and a few of payload, which should well characterize a session. When trimming all sessions into a uniform length, 0×00 is appended to complement it to n bytes if the size of a session is shorter than n bytes. After trimming, each session will be converted to the input format of the inference model for training and testing. Once the model has been trained, to make the NCS be able to install the training model for inference, we should configure the training model for compiling an inference model.

After generating an inference model for an NCS, we can run the inference model within an NCS for online inference. When a P4 switch receives the network traffic of a full session, we should do preprocessing to generate an input data for NCS

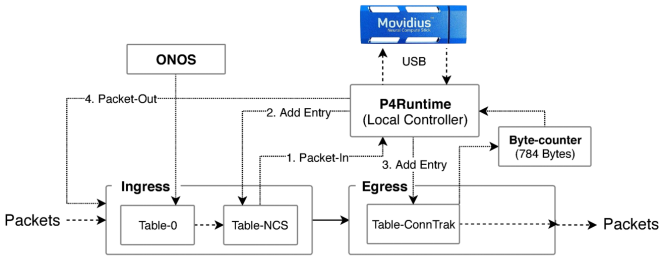

Fig. 5: P4 filtering rule

to perform inference. After getting the result of inference, the manager can consider to send the whole result or just only a notification of anomaly to the local controller. Based on the inference results, the controller can adapt its strategy by modifying the forwarding rules of switches or banning some misbehavior flows.

B. Flow Filtering

To capture only the first n bytes for inference, we leverage the configurable P4 switches to direct only the first n bytes to P4Runtime for further process. By doing this, we can significantly reduce the load of forwarding data among P4 switch and P4Runtime. Figure 5 plots the process of the proposed P4 filtering rule. In the beginning, Tofino will trigger a packet-in event and send the packet to the local controller during the ingress stage. Besides, the local controller will also buffer the packets. The local controller will add an entry into the NCS table and the other table for connection tracking. The byte-counter keeps tracking the number of bytes according to the table in connection tracking. The reason why the byte-counter is tracking the number of bytes according to the table in connection tracking is to ensure that the number of bytes can accumulate from the first packet of a session without missing the first packet when table-miss occurs in the NCS table. After collecting the first n bytes of a session, Tofino will notify the local controller that the queue is ready for inference.

C. In-Switch Inference

Upon receiving the notification, our system transforms the data in queue into the input data format of a model and sends to the NCS. The NCS then outputs the inference result and returns the result back to the local controller. The local controller will add an entry into the NCS table and the table for connection tracking according to the inference result from the NCS. By doing this, we can promptly detect misbehavior or network properties via switches and configure rules immediately to react to network changes.

IV. Evaluation

We implement a prototype of INI to evaluate its performance. In our implementation, an Intel Movidius Neural Compute Stick 1 (Intel NCS-1) is connected to a P4 Wedge100BF-32X switch via a USB port. We compare our INI with a legacy architecture, where the P4 switch is connected to a server with two GeForce GTX 1080 Ti GPU for remote inference. To verify the effectiveness of our design, we use malware classification as an application. We use a real data set, DeepTrack (USTC-TFC2016) [11], which includes 791,615 packets, to train a CNN-based classification model. We define packets with the same five-tuple as a flow, and the data set includes 288,614 flows. The CNN model contains two convolutional layers, two maxpool layers and two fully connected layers. The model is trained with 20,000 epochs. Each session is transformed to a gray-scale 28×28 image as the input of the inference model. Hence, we let each switch forward the first 784 bytes of a flow to the P4Runtime for inference in NCS.

Metrics (ms)	Legacy	INI
Transmission per packet	39.53	0.378
Image transformation	0.35	0.38
Inference	3.78	8.56
Total	43.66	9.318

TABLE I: Inference Execution Time

Execution Time of In-Switch Inference: We partition our INI framework into three steps: packet forwarding (transmission), transformation from packets to images, and inference. We measure the time required by each phase in Table I for the legacy GPU architecture and our INI, respectively. To measure the transmission time between the P4 switch and P4Runtime (or GPU server), we transmit the real traffic received by Tofino to P4Runtime via Tofino ingress or the remote GPU server through a 100 Mbps link, and calculate the average transmission time of each packet. Note that the legacy architecture requires each switch to forward the packets to a backend server. To emulate this scenario, we generate additional 9 connections that also send a file to the GPU server at the same time. As multiple switches share a bottleneck link connecting to the GPU server, the transmission time would be fairly high. However, INI only needs to redirect the packets from the P4 switch to P4Runtime. Hence, the data forwarding path is extremely short, as a result reducing the transmission time significantly.

As for image conversion, since both the legacy and INI architecture leverages computing power, the process time is similar. Finally, to measure the inference time, we perform inference for all the flows and find the aggregated time required to finish all the inference tasks. The table then shows the inference time per flow on average. As the NCS is a less capable device, its inference speed is slightly worse than GPU. Also, our result overestimates the performance of GPU since we simply calculate the average time required by a single inference request without considering the queueing delay. However, a GPU server has to serve many switches and may buffer lots of requests in its queue. In other words, the actual inference time, including the queueing delay, could be longer in the legacy architecture. After summing up all the processing time, we can see that INI reduces the overall execution time significantly, as compared to the legacy architecture, showing the effectiveness of our intra-network inference design.

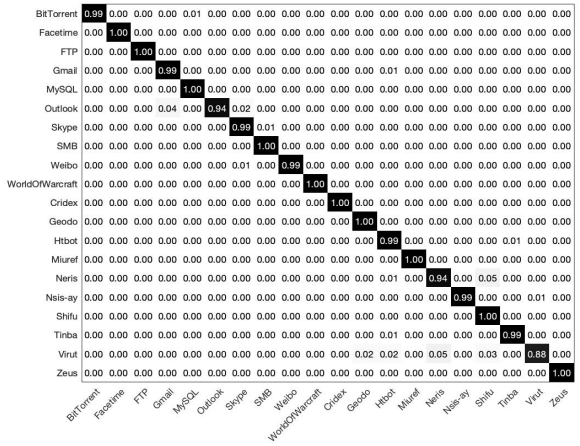

Fig. 6: Malware Classification Accuracy

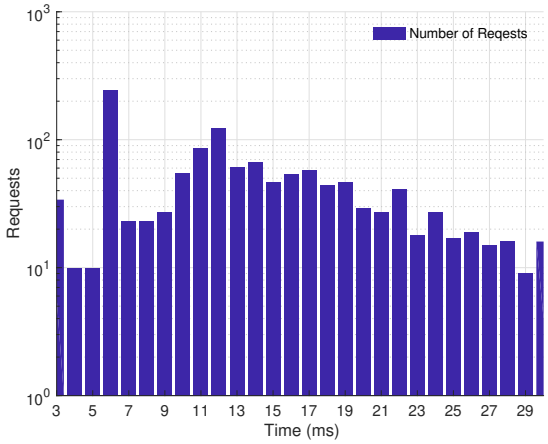

Fig. 7: Inference Request Interval

Inference accuracy: The trained model is a multi-class classification model, which can classify a flow into one of the potential applications (normal or malware). We reserve 80% of the flows to train the CNN model and use the remaining 20% of the flows to test the model accuracy. The prediction results is shown as a confusion matrix summarized in Figure 6. The figure shows that most of the classes can be accurately classified with a prediction ratio higher than 94%. Though the processing speed of the NCS is slightly worse than the GPU server, it can still achieve an accurate prediction performance, showing its potential of realizing intra-network machine learning.

Inference request arrival rate: We finally examine whether the inference capability of NCS is sufficiently high for real-time traffic. To check what is the arrival rate of inference sessions, we extract the timestamp of each packet and find the timestamp of the packet that includes the 784-th byte of the flow. When this packet arrives, it means that a new inference request will be sent to P4Runtime. That is, the timestamp of the packet including the 784-th byte is exactly the timestamp of an inference request. We then count the number of inference requests arrived every millisecond.

Figure 7 plots the number of inference requests every millisecond over time. The figure shows that, in most of time, there exist more than 10 requests every millisecond. However, each inference should be processed using 8.56 ms, on average, as shown in Table I. This result verifies that the arrival rate of inference requests is way more higher than the service rate of an NCS (actually the service rate of a GPU as well). That is, it is not possible to just rely on a single NCS to handle all the inference requests in a network. It is hence worth study about how to leverage multiple NCS-equipped switches to share the inference load of a network in the future.

V. Conclusion

In this work, we presented an intra-network inference architecture, called Intra-Network Inference. We combine programmable switches with a recently-released component, i.e., neural compute stick (NCS), to enable switches to performance local inference. We develop filtering rules in the switch and communication channels among the switch and P4Runtime to realize in-network inference. By doing this, our architecture avoids the heavy load of data forwarding among data and control planes and further enable real-time network management inside the network. We implement a prototype of INI to measure the execution time required by each processing step and point out some potential future research directions.

References

[1] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *ACM SIGCOMM*, 2018.

[2] T.-Y. Mu, A. Al-Fuqaha, K. Shuaib, F. M. Sallabi, and J. Qadir, "SDN flow entry management using reinforcement learning," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, no. 2, Nov. 2018.

[3] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *International Conference on Information Networking (ICOIN)*, Jan 2017.

[4] Y. Huang, W. Shih, and J. Huang, "A classification-based elephant flow detection method using application round on SDN environments," in *19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sep. 2017.

[5] Peng Xiao, Wenyu Qu, Heng Qi, Yujie Xu, and Zhiyang Li, "An efficient elephant flow detection with cost-sensitive in SDN," in *1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, March 2015.

[6] A. AlGhadhban and B. Shihada, "FLight: A fast and lightweight elephant-flow detection mechanism," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2018.

[7] S. C. Madanapalli, M. Lyu, H. Kumar, H. H. Gharakheili, and V. Sivaraman, "Real-time detection, isolation and monitoring of elephant flows using commodity SDN system," in *IEEE/IFIP Network Operations and Management Symposium*, April 2018.

[8] B. Yamansavascilar, M. A. Guvensan, A. G. Yavuz, and M. E. Karsligil, "Application identification via network traffic classification," in *International Conference on Computing, Networking and Communications (ICNC)*, Jan 2017.

[9] A. Tongaonkar, R. Keralapura, and A. Nucci, "Challenges in network application identification," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2012.

[10] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[11] "Deeptraffic dataset." [Online]. Available: https://github.com/echowei/DeepTraffic/tree/master/1.malware_traffic_classification