When NFV Meets ANN: Rethinking Elastic Scaling for ANN-based NFs

Menghao Zhang⋆, Jiasong Bai⋆, Guanyu Li⋆, Zili Meng⋆, Hongda Li†, Hongxin Hu†, Mingwei Xu⋆
⋆Tsinghua University †Clemson University

***Abstract*—Network Function Virtualization (NFV) provides middleboxes with substantial elasticity from a system level, and Artificial Neural Network (ANN) empowers middleboxes with great intelligence from an algorithm-level perspective. However, when ANN-based Network Functions (NFs) want to take advantage of the elasticity of NFV, our study finds that huge gaps exist between the existing approaches and the ideal goals for the elasticity control of ANN-based NFs. By revealing the key differences between ANN-based NFs and traditional NFs, we propose LEGO, an innovative framework that provides systematic mechanisms for traffic splitting, instance partition and runtime management to enable correct and efficient scaling of ANN-based NFs. Preliminary implementation and evaluation demonstrate the feasibility and effectiveness of the LEGO system. The major purpose of this paper is to highlight these challenges and sketch out a new roadmap towards ANN-based NFV paradigm.**

I. INTRODUCTION

Network Functions (NFs), or middleboxes, have become a crucial part of today's Internet. Complementary to the forwarding functions provided by routers, middleboxes play an important role to ensure security (e.g., firewall, intrusion prevention/detection system), improve performance (e.g., load balancing, WAN optimizer) and provide other novel network functionalities [1], [2].

While traditional NFs are deployed as proprietary monolithic software running on dedicated hardware, network operators are moving towards Network Function Virtualization (NFV), in which middlebox functionalities are moved out of dedicated physical boxes into virtual appliances that can be run on commodity servers. Since it emerges, NFV has quickly gotten significant attention from both academia and industry, and hundreds of industry participants are planning to deploy, or have already deployed it to achieve high elasticity and to reduce management expenses.

Orthogonal and complementary to the system-level solutions like NFV, which virtualizes the softwarized NFs, another trend in improving NFs origins more from an algorithm-level perspective. With the wide applications and successes of Artificial Neural Network (ANN) [3] in computer vision [4], natural language processing [5], [6], and voice recognition [7] recently, the networking community has also started to embrace ANN to tackle some thorny problems that long exist in networking. In particular, some researchers begin to adopt ANN to compose sophisticated NFs to achieve advanced packet processing to satisfy the performance and security goals [8], [9], [10], [11], [12], [13]. As reported previously [14], [1], traditional NFs suffer from high infrastructure and maintenance costs, requiring significant manual efforts and operator expertise to make effective decisions. For example, in an Intrusion Detection System (IDS), new attacks emerge every day, and the operators need to update the signature inspection rules all the time [8]. In a network-wide load balancing, hand-crafted heuristics for varying traffic load, flow size distribution, traffic concentration are chosen intuitively to make traffic optimization decisions. Even in this way, the decisions are always suboptimal and lead to overall bandwidth waste [9]. ANN provides a good opportunity to tackle these problems, since it is good at *learning sophisticated non-linear concepts* and *making near-optimal decisions under complex, uncertain environments* [3]. These characteristics of ANN reduce management costs and operational expenses significantly.

978-1-7281-2700-2/19/$31.00 2019 © IEEE

Despite the substantial benefits of these two separated trends for improving NFs, it is still unknown whether ANN-based NFs could take advantage of the *elasticity* property provided by NFV. Our study reveals that there is a huge gap between the existing solutions for NFV and adopting them to ANN-based NFs, and several key challenges must be carefully addressed to fully achieve the vision for elastic ANN-based NFs. The goal for elastic scaling of ANN-based NFs is to satisfy the tight Service Level Agreement (SLAs) and minimize the operation costs under frequently varied traffic volume. As mentioned in previous works [15], [16], [17], the scaling of NFs must carefully fulfill the properties of *correctness* and *efficiency*. However, ANN-based NFs are much different from traditional rule-based NFs, and they *have a monolithic, unseparated NF state* and *are greedy on computation resources*. These unique characteristics make traditional NF scaling approaches (e.g., Split/Merge [15], OpenNF [16], S6 [17]) either incorrect or inefficient (details in §II-B). A more advanced coordination mechanism among networks, processing and NF states is highly desired to take full advantage of the benefit of ANN-based NFs, which should simultaneously accomplish correctness and efficiency properties.

To bridge the above gaps, in this paper, we propose LEGO, a novel framework that provides coordinated control for NF instances and network forwarding to allow efficient and correct scaling of ANN-based NFs. LEGO jointly manages the traffic splitting and NF instance organization to tackle the problems mentioned above, with three essential elements. First, LEGO proposes an effective traffic splitting mechanism combined with the inherent neural network structure to guarantee correctness. Second, LEGO proposes to break up the ANN-based NF instances into smaller *bricks*, which could be moved and replicated independently. This is inspired by the recent trend towards micro-services [18], which could greatly reduce the composition complexity of ANN-based NFs and potentially achieve high resource efficiency. Third, LEGO proposes to continuously monitor the resource utilization of each brick with the LEGO controller, and replicate/merge any overloaded/underloaded bricks, which could further achieve resource efficiency for each machine and across machines.

As a proof of concept, we leverage a state-of-the-art ANN-based IDS, Kitsune [8], and apply our approach to carry out the scaling experiments. Our preliminary results demonstrate that our approach is highly effective to achieve the efficiency and correctness of the elastic scaling for the ANN-based IDS. Looking forward, there are apparently a lot of questions remaining to be answered. For instance, can we adapt the LEGO framework to other ANN-based NFs? How to enhance the performance and fault tolerance of ANN-based NFs? Can we further reduce the management expense of ANN-based NFs? These issues require cooperations among researchers from diverse fields to move towards a new ANN-based NFV paradigm.

In summary, the contributions of this paper are as follows:

- We study the structure of ANN-based NFs and identify the key differences from traditional NFs (§II).
- We propose LEGO, an ANN-based NF scaling framework, with three essential elements to obtain the benefit of efficient and correct scaling (§III).
- We implement a proof-of-concept prototype with a typical ANN-based NF instance, KitSune, and conduct the preliminary experiment to show the effectiveness of LEGO (§IV).

Finally, we conclude this paper with our ongoing explorations in Section V, and hope that this paper could act as a catalyst to spark the debate on this impending field.

II. BACKGROUND AND MOTIVATION

A. Background on ANN-based NFs

Inspired by biological neural networks, Artificial Neural Network (ANN) exploits many layers of non-linear information processing for supervised or unsupervised feature extraction [19]. Each layer is a collection of processing units called neurons, and each neuron is followed by an activation function, which is used to generate the neuron's output. And the interconnection between neurons of two layers is associated with parameters to determine how much one neuron could affect another. With these weighted interconnections, the output of each neuron is transferred to all neurons in the next layer to activate neurons of the next layer iteratively. Before being used to infer new samples, ANN should be trained first. In particular, training ANN is an optimization procedure, where the weight parameters of each interconnection should learn the most appropriate value through the training dataset, aiming to capture the complex patterns of non-linear relationships between the inputs and the outputs. The often-used learning algorithm in ANN is back propagation algorithm, in which training samples are forwarded and the output of the network is compared with the desired target. Then the weight parameters would be tuned according to this difference.

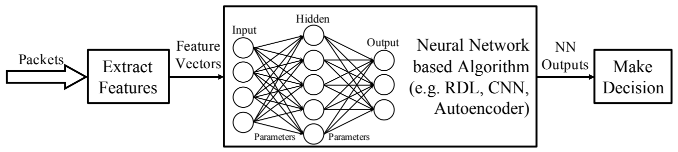

Figure 1: A General Architecture of ANN-based NF.

To reduce management costs and make near-optimal decisions, recently there is a growing trend to leverage ANN to support various NFs [8], [9], [10], [20], [11], [12], [13]. Kitsune [8] employs an ensemble of neural networks called Autoencoders to implement a plug-and-play IDS, which can learn to detect attacks without supervision and detect new emerging attacks without the deep involvements of network operators. Similarly, based on Autoencoder and Convolutional Neural Network (CNN), BoTShark [13] is able to detect centralized or P2P botnets effectively without operators' expertise. To automate traffic optimization process (e.g., load balancing) in the data center, AuTO [9] develops a two-level Deep Reinforcement Learning (DRL) system, which achieves significant performance improvement compared to standard human-crafted heuristics. All these examples achieve pretty promising results, which demonstrate the substantial advantages brought by ANN.

From these typical ANN-based NFs, we summarize the general architecture of ANN-based NFs, as shown in Figure 1. Generally, the first step of ANN-based NFs is to extract a series of feature vectors from packet batching. Then ANN-based NFs would assign these feature vectors to the input layer and utilize different ANN (e.g. Autoencoder, CNN, DRL, RNN, etc.) to approximate these data patterns in unsupervised or supervised manners. Finally, a decision is made based on the outputs of ANN to obtain the final results, such as intrusion alerts in an ANN-based IDS or selected forwarding paths in an ANN-based load balancing. Actually, this general architecture is perfectly applicable for all examples above. For Kitsune [8], it first adopts incremental statistics maintained over a damped window to extract the features of traffic, then selects an ensemble of Autoencoders as the fundamental neural network with the root mean squared error (RMSE) computed as its outputs, and finally decides whether an alert is produced based on the RMSE values. In BotShark [13], it first employs specific tools to extract NetFlow which record information of connections, then applies stacked Autoencoders to learn implicit features from NetFlows and utilizes CNNs to implement a classifier with all features as input, and finally selects Softmax to make a decision based on outputs of CNNs. In AuTO [9], the monitoring module first collects a large amount of flow information, then DRL module learns the characteristics of traffic with normalized throughput as rewards, and finally DRL agent determines rates, routes, and priorities for long flows automatically. Obviously, AuTO also satisfies such a general architecture.

B. Observation and Motivation

To satisfy the tight SLAs and minimize operation costs under frequently changing traffic volume, ANN-based NFs also need to create or destruct their NF instances dynamically, as traditional NFV instances do [15], [16], [17]. However, to achieve this in a correct and efficient way is non-trivial, and our study reveals that a more advanced structure for NF instance along with a more superior network forwarding technique is required, which is beyond the capability that state-of-the-art approaches (e.g., Split/Merge [15], OpenNF [16]) can provide. The rationales behind this gap lie in the unique characteristics of ANN-based NFs, as summarized in the following points:

First, ANN-based NFs replace classical separated rule-based states with monolithic numerical representation. Such systems learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. Traditional NFs maintain the network states (including static and dynamic state as illustrated in Split/Merge [15], OpenNF [16], StatelessNF [21] and S6 [17]) as separated rules. Although there is dependence between different rules, the correlation among these rules is limited. As a result, when we want to create/destruct an instance, we can easily replicate/merge the state to the instance and steer the traffic accordingly. In contrast, ANN-based NFs implicitly encode all these rules with weighted parameters, and they behave as a monolithic and identical part, making the state-oriented traffic splitting a challenging problem. If this problem is not properly dealt, the correctness cannot be guaranteed. For example, an alert raised in one ANN-based IDS instance may be lost if the traffic is split to multiple instances carelessly. Therefore, an advanced approach is in dire need to split the traffic correctly and effectively.

Second, ANN-based NFs are more sophisticated than traditional NFs, and they usually require more computation resources to conduct packet batching, traffic feature extracting, feature vector mapping, training and inference[1], and context-related decision making. According to our experiments on two typical IDSes, when allocated with one CPU core, Kitsune, an ANN-based IDS, can only process hundreds of packets per second, while Snort, a classic signature-based IDS, is able to process tens of thousands of packets in one second. This implies the resource-intensive characteristic of ANN-based NFs. As a result, if ANN-based NFs are scaled monolithically as traditional NFs, the *scaling procedure would be slow* because of tremendous resource allocation, and many *fractional resources are unusable* by the ANN-based NF instances. As a result, an effective approach is urgently needed to reconstruct the structure of ANN-based NFs to allow efficient scaling.

These characteristics mentioned above make the traditional NF scaling approaches no longer applicable, and a new framework customized for ANN-based NFs is needed to carry out the efficient and correct scaling to fully achieve the benefit of ANN-based NF virtualization.

[1]Even in most cases, training task is conducted offline and beforehand. Inference task must be finished online and timely.

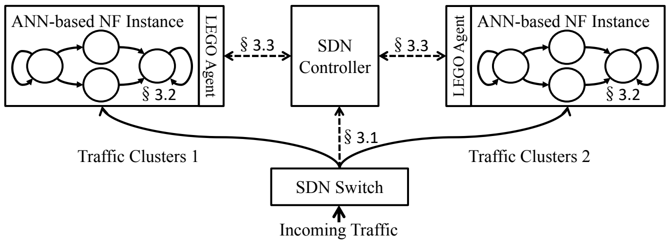

Figure 2: LEGO Architecture.

III. Our Approach: Lego

In this section, we propose a coordination framework, namely LEGO, to help efficiently and correctly scale the ANN-based NFs. The overall architecture of LEGO is shown in Figure 2. In general, LEGO consists of three key components. First, LEGO introduces an effective traffic splitting mechanism combined with the inherent neural network structure to guarantee correctness (§III-A). Second, LEGO proposes to break up the ANN-based NF instances into smaller NF bricks to achieve independent and fast replication, which could reduce the composition complexity of ANN-based NFs and pave the way for high resource efficiency (§III-B). Third, LEGO continuously monitors the resource utilization of each brick with the LEGO controller, and replicates/merges any brick that is overloaded/underloaded, which could further achieve resource efficiency for machine clusters (§III-C).

A. Traffic Splitting

To scale in/out ANN-based NFs, we first need to split the traffic across multiple NF instances. An essential requirement for traffic splitting is correctness, i.e., the output of the NF instances before and after the traffic splitting should be the same. For traditional NFs, traffic can be split directly at an explicit granularity (e.g., flow) with certain NF state replicated to the new instances. However, this is not applicable for ANN-based NFs, since NF states are encoded into monolithic, unseparated weighted parameters. As a result, traffic cannot be split explicitly as traditional NFs. If not dealt carefully, this may lose essential features for feature extractor and further compromise detection soundness. To see why, consider a scenario where an ANN-based IDS instance, Kitsune, is deployed at the Internet of Things (IoT) gateway, which can effectively detect the port scanning attacks. When the Kitsune instance is overloaded and must be scaled out to satisfy the SLAs, we first create a new Kitsune instance with weighted parameters already loaded. Then we steer half of the traffic to the new instance to conduct the inference. Because of the traffic splitting, the feature extractor cannot retrieve the complete feature vectors as origin, and the RSME values outputted from the ANN may be small than the original anomaly threshold.

To address this problem, we first revisit the basic processing procedure of ANN-based NFs. As shown in Figure 1, raw packets are first processed by feature extractor in batch, to

Algorithm 1: Traffic Splitting Granularity Selection.

```
Input: Granularity Set $\mathcal{D} = \{header\}$.
Output: Selected Granularity.
1 foreach $header \in \mathcal{D}$ do
2     $header.indegree = 0$
3 foreach $header1 \in \mathcal{D}$ do
4     foreach $header2 \in \mathcal{D}/\{header1\}$ do
5         if $header1 \subset header2$ then
6             $header1.indegree$ += 1
7 return all $header$ where $header.indegree$ == 0
```

retrieve the feature vectors that will be the input of ANN. To make the output of NF instances before and after traffic splitting the same, the input of the ANN should also be the same, which requires our traffic splitting mechanism to respect the scheme of the feature extractor. Generally speaking, the feature extractor is used to capture the context and purpose of each packet traversing the network, and can be regarded as a mapping from batched packets into a set of feature vectors, which are usually a composition of the characteristics of the batched packets, including the packet protocol, packet size, packet number in a time window, packet jitter and so on. We observe that although the characteristics of the batched packets are wide and various, the only factor that affects the traffic splitting is the packet protocol, since other factors are orthogonal to the packet protocol property. As a result, if our traffic splitting mechanism respects the packet protocol property of the feature extractor, the input of ANN will remain the same. Taking Kitsune as an example, it extracts a behavior snapshot of hosts and protocols communicated with the given packets, which consists of 115 traffic statistics summarizing all of the traffic originating from this packet's source MAC and IP address (SrcMAC-SrcIP), this packet's source IP (SrcIP), this packet's source and destination IP (SrcIP-DstIP), and this packet's source and destination TCP/UDP sockets (Socket). In this context, the packet protocol property consists of four groups, SrcMAC-IP, SrcIP, SrcIP-DstIP and Socket. In general, there are two kind of relationships between these groups: inclusion $\subset$, i.e., $Group_1 \subset Group_2$, and overlapping $\cap$, i.e., $(Group_1 \cap Group_2 \neq \emptyset) \wedge (Group_1 \cap Group_2 \neq Group_1) \wedge (Group_1 \cap Group_2 \neq Group_2)$. For example, SrcMAC-IP $\subset$ SrcIP, and SrcIP-DstIP $\cap$ SrcMAC-IP. To select the ideal traffic splitting granularity, we design an algorithm as shown in Algorithm 1. The input of this algorithm is the protocol groups, and the output is the selected splitting granularity, which can be categorized into two cases:

- One result, which means the incoming traffic can be split based on certain granularity. The feature extractor of Kitsune falls into this category, so the traffic can be split with SrcIP granularity into different Kitsune instances.
- Two or more results, which indicates that the incoming traffic cannot be explicitly split. We have to replicate some relevant packet to the corresponding NF instances to guarantee correctness, or trade-off some correctness for overhead. We leave the detailed exploration of this part as our future work.

B. ANN-based NF Partition

Monolithically provisioned ANN-based NFs are inefficient in scaling and resource allocation, since ANN-based NFs require more computational and memory resources to conduct packet batching, traffic feature extracting, feature vector mapping, training and inference, and context-related decision making. Scaling a monolithic ANN-based NF instance is slow because of tremendous memory initiation for the new instance, incurring significant latency of the traffic. Besides, as ANN-based NFs consume more resources than traditional NFs, monolithic provisioning of ANN-based NFs results in more waste of fractional resources in the environment. For example, assume an ANN-based NF requires 8 CPU cores to run. If the NF is provisioned with a single virtual machine, one should assign 8 CPU cores to run the virtual machine. In this case, if the physical machine only has 4 CPU cores available, there is no way for the ANN-based NF to make use of these 4 cores. Therefore, breaking a monolithic ANN-based NFs down into pieces is critical to enable elastic scaling for ANN-based NFs, which is much more necessary than traditional NFs. We choose to partition a monolithic ANN-based NFs into several small units, which are named as *bricks* in our paper.

Partition Principle. Partitioning a monolithic NF instance into bricks is non-trivial, and it requires an intricate balance: if a brick contains too little functionalities, there would be too many inter-bricks communications, leading to high overhead and considerable latency; if a brick is too large, the goals for fine-grained composition and coordination of NF bricks could not be reached easily, which would fail to satisfy the requirements of resource efficiency. Therefore, we present our two rules of thumb for partitioning. First, the cost incurred by brick order book-keeping and communications between bricks should be much less than the cost of replicating a new brick. Second, each brick should have as less independence as possible to the other bricks. Combined with the unique characteristics of ANN-based NFs, we present some preliminary results as a useful start point. As shown in Table I, we categorize these NF bricks into three types: Feature Extractor as the preprocessing for ANN, different types of ANNs, and Decision Maker as the processing for sample results or environments, also for output representation. We are seeking for more ANN-based NF instances and developing more NF bricks to make our system more general and effective. Note that in the current context we do not further partition the ANN into small bricks based on different layers. This is because the ANN in today's NFs is not so deep and can be executed very quickly, for example, the autoencoders in Kitsune have only three layers, the DNN in AuTo has only ten layers. The deep reason behind this may be that the networked systems have a high requirement for ANN's execution speed, as networking is an online system essentially.

Brick Component and Structure. Brick is the basic unit for functionalities, and it must have the following components: (1) An identifier to identify its type and its serial number, which is exposed to the LEGO controller for easier manage-

Table I: Partial list of NF Bricks.

Brick Name	*Description*	*Category*
Packet Batching	Read a batch of packets in memory	Feature Extractor
Feature Quantization	Select a set of metadatas and Quantize them into feature vectors	Feature Extractor
Feature Mapping	Map the features into the input of ANN	Feature Extractor
Supervised Learning Network	The network learns the desired output for a given input or pattern	ANN
Unsupervised Learning Network	The network learns without specifying the desired output	ANN
Hybrid Learning Network	The network combining the two properties above	ANN
Rewarding Functions	Map the environment rewarding into the output of ANN	Decision Maker
Sample Result Quantization	Map the sample result into the output of ANN	Decision Maker

ment and maintenance. (2) A routing table, which steers the traffic to the next-hop bricks. (3) A northbound interface to receive and report necessary information from/to the LEGO controller. (4) A west-east interface to communicate with its replica bricks. Inter-bricks communication takes place via Inter-Process Communication (IPC) in the same server, and could extend to Remote Procedure Call (RPC) if the resource on one machine is inadequate. The controller could forge some faked packet headers to steer some packets from one server to another [22]. Each brick adopts a stateless programming model, i.e., the state could be logically separated from the processing logic. Since the state and the processing logic are in the same machine in most cases, this programming model will not degrade the performance for processing but provide substantial elasticity.

C. Runtime Management

LEGO has a controller, which is responsible for allocating the resource and scheduling the NF bricks at runtime. The resource allocation and brick scheduling mechanism are NF-specific, i.e., it should be adapted to the structure of that specific ANN-based NF. Scheduling decision includes the initial and subsequent placement of NF bricks (bricks graphs), and the assignment of traffic to NF bricks. The goal of the LEGO controller is to monitor the resource consumption of each NF brick and balance the loads across NF brick replicas to efficiently satisfy the tight SLAs. During the bricks graph construction procedure, the controller first accepts the overall SLA requirements from the network tenants, then it obtains the brick-level deadlines by dividing the end-to-end level constraints among the bricks along a path of the graph, proportionally to their computation costs.

Brick Cost Estimation. To make the resource allocation decision, we first need to know the resource cost of each brick, which mainly include (1) the amount of computation and memory resource to process an batch of input data, (2) the number of output data to be transmitted to a downstream brick, and the corresponding bandwidth. Since the resource costs of each brick under the same input data are relatively stable, we could resort to an analysis of source code (e.g., using existing timing analysis tools [23]) or pre-execution under real-world traces. Sometime the cost may change due to runtime dynamic [24], we could update the cost estimation periodically.

Runtime Monitoring and Brick Replication. At runtime, the LEGO controller detects bottlenecks in a brick graph by monitoring a set of critical metrics of each brick, including the input and output queues, the CPU load, memory and I/O utilization, and the load of the switches. When detecting a bottleneck brick, the controller replicates a replica, re-allocates the resource, re-assigns the traffic, and updates the routing table. This procedure first takes place in one machine, since the bricks in one machine could share the same memory and achieve high resource efficiency. If the resource of one machine is exhausted, there are two approaches we could take. First, allocate a complete brick graph in a new machine with the complete weighted parameters and steer the traffic accordingly. Second, allocate only the overloaded bricks in the new machine, and steer the inter-bricks traffic to the new bricks. This could achieve high resource efficiency but at the expense of the hard merging mechanism at the downstream bricks. The choice of these two designs should consider the physical locations of the NF clusters. We leave the detailed replicating/merging mechanisms as our future work.

Table II: Traffic Splitting with Different Methods.

Methods	**Original**	**SrcMAC-SrcIP**	**SrcIP**	**SrcIP-DstIP**	**Socket**
RMSE	10.117	10.082	10.083	2.964	2.973

IV. CASE STUDY

To validate the effectiveness of our LEGO framework, we developed a proof-of-concept prototype based on Kitsune [8]. We partition Kitsune into bricks as the manners mentioned in §III-B, and connect these bricks into a brick graph. We implement an initial version of the LEGO controller adapted from POX [25]. Our experiments are conducted in two directly connected servers, one for deploying Kitsune or LEGO, and the other for replaying real-world traffics.

To explore the effectiveness of the proposed traffic splitting scheme in LEGO, we split Mirai traffic [26] into three parts with different granularities, replay these parts respectively on the original Kitsune and measure the average RMSE of all packets as the metric of correctness. As shown in Table II, when traffic is split with SrcIP granularity (the output of our algorithm), it will produce the largest RMSE value and generate the closest results with the original single-instance case. We can also see that SrcMAC-SrcIP produces a similar RSME value with SrcIP, because the distribution of traffic splitting under these two circumstances are almost the same. While for other cases, the RSME values are much smaller, which indicates that some anomalies are ignored because of careless traffic splitting schemes. To conclude, traffic split-

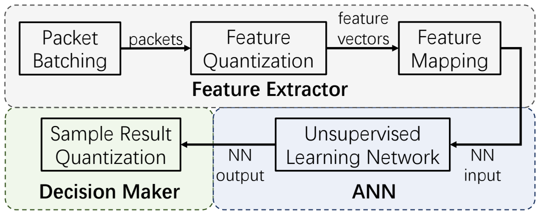

Figure 3: Kitsune's LEGO Brick Graph.

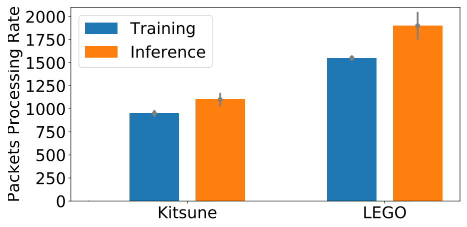

Figure 4: Throughput Improvement by LEGO Brick Partition and Placement.

ting granularity selection algorithm in LEGO guarantees the correctness for traffic splitting when scaling in/out the ANN-based IDS.

To demonstrate the efficiency of LEGO for resource utilization, we reconstruct Kitsune with LEGO bricks, as shown in Figure 3. We assign one separated CPU core for each brick of the LEGO Kitsune while 5 cores are allocated to the original Kitsune to guarantee fairness. When the LEGO Kitsune is running, the LEGO controller monitors the resource occupation of each brick, and detects that the bottleneck exists at *Unsupervised Learning Network (Autoencoder)* brick where it takes almost 100% CPU core. As a consequence, the LEGO controller replicates a replica and re-assigns a new CPU core for the Autoencoder. In contrast, for original Kitsune, with the computation-intensive Autoencoder module becoming the bottleneck, it could not make full use of CPU cores in one machine and is limited by the processing capacity of the Autoencoder. As shown in Figure 4, LEGO nearly double the throughput within one machine. In conclusion, the LEGO Kitsune is able to achieve higher resource utilization, benefiting from its NF partitioning and runtime management.

V. CONCLUSION AND FUTURE WORK

In this paper, we identify the key challenges to enable correct and efficient scaling of ANN-based NFs, and sketch the vision of LEGO, an innovative framework to provide advanced mechanisms for traffic splitting, instance partition and runtime management to achieve such a goal. We further apply our methodology to a state-of-the-art ANN-based IDS, Kitsune, to demonstrate the effectiveness of our approach.

Nevertheless, our current design, prototype and experimental results are in a very preliminary stage, which leaves a lot of explorations to continue. In our ongoing work, we are planning to investigate the cases where traffic cannot be split perfectly and how to achieve correctness with the minimal replication packets, partition ANN into different layers to get small bricks if necessary, apply our approach to more ANN-based NFs to demonstrate the generality of LEGO, see how to further enhance the performance and fault tolerance of ANN-based NFs, and consider more complex scenarios.

VI. ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable comments. Menghao, Jiasong, Guanyu and Zili are also sincerely graceful for their former Ph.D. advisor, Jun Bi from Tsinghua University, for his strong support. This work is supported by the National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61872426).

REFERENCES

[1] J. Sherry and et al., "Making middleboxes someone else's problem: network processing as a cloud service," *SIGCOMM*, vol. 42, no. 4, pp. 13–24, 2012.
[2] Z. Wang and et al., "An untold story of middleboxes in cellular networks," in *SIGCOMM*, vol. 41, no. 4. ACM, 2011, pp. 374–385.
[3] Wikipedia, "Artificial neural network," 2018, https://en.wikipedia.org/wiki/Artificial_neural_network.
[4] A. Krizhevsky and et al., "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.
[5] A. Vaswani and et al., "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.
[6] J. Devlin and et al., "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[7] G. Hinton and et al., "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.
[8] Y. Mirsky and et al., "Kitsune: an ensemble of autoencoders for online network intrusion detection," *NDSS*, 2018.
[9] L. Chen and et al., "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," *SIGCOMM*, 2018.
[10] M. Cheng and et al., "Ms-lstm: A multi-scale lstm model for bgp anomaly detection," in *ICNP*. IEEE, 2016, pp. 1–6.
[11] L. He and et al., "Vtc: Machine learning based traffic classification as a virtual network function," in *SDN & NFV*. ACM, 2016, pp. 53–56.
[12] R. Doshi and et al., "Machine learning ddos detection for consumer internet of things devices," *arXiv preprint arXiv:1804.04159*, 2018.
[13] S. Homayoun and et al., "Botshark: A deep learning approach for botnet traffic detection," *Cyber Threat Intelligence*, pp. 137–153, 2018.
[14] A. Bremler-Barr and et al., "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *SIGCOMM*. ACM, 2016, pp. 511–524.
[15] S. Rajagopalan and et al., "Split/merge: System support for elastic execution in virtual middleboxes." in *NSDI*, vol. 13, 2013, pp. 227–240.
[16] Gember-Jacobson and et al., "Opennf: Enabling innovation in network function control," in *SIGCOMM*, vol. 44, no. 4. ACM, 2014, pp. 163–174.
[17] S. Woo and et al., "Elastic scaling of stateful network functions," in *NSDI*. USENIX Association, 2018.
[18] ThoughtWorks, "Microservices: Lessons from the frontline," 2018, https://www.thoughtworks.com/insights/blog/microservices-lessons-frontline.
[19] L. Deng and et al., "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.
[20] J. Saxe and et al., "A deep learning approach to fast, format-agnostic detection of malicious web content," *arXiv preprint arXiv:1804.05020*, 2018.
[21] M. Kablan and et al., "Stateless network functions: Breaking the tight coupling of state and processing." in *NSDI*, 2017, pp. 97–112.
[22] S. Vissicchio and et al., "Central control over distributed routing," in *CCR*, vol. 45, no. 4. ACM, 2015, pp. 43–56.
[23] S. Chattopadhyay and et al., "A unified wcet analysis framework for multicore platforms," *TECS*, vol. 13, no. 4s, p. 124, 2014.
[24] T. Li and el al., "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC*. IEEE, 2007, pp. 1–11.
[25] J. Mccauley, "Pox: A python-based openflow controller," 2014.
[26] Y. Meidan and et al., "N-baiot: Network-based detection of iot botnet attacks using deep autoencoders," 2018.