5G CITY

Grant Agreement No.761508 5GCITY/H2020-ICT-2016-2017/H2020-ICT-2016-2

D4.3: Location-aware Machine Learning Mechanisms

Dissemination Level		
X	PU:	Public
☐	PP:	Restricted to other programme participants (including the Commission Services)
☐	RE:	Restricted to a group specified by the consortium (including the Commission
☐	CO:	Confidential, only for members of the consortium (including the Commission Services)

Grant Agreement no: **761508**	Project Acronym: **5G CITY**	Project title: **5G CITY**

Lead Beneficiary:	Document version: **V1.0**

Work package: WP4

Deliverable title: Location-aware Machine Learning Mechanisms

Start date of the project: 01/06/2017 (duration 30 months)	Contractual delivery date: M28 Month	Actual delivery date: 01-10-2019

Editor name: Roberto Bifulco (NEC)

List of Contributors

Participant	Short Name	Contributor
NEC Europe	NEC	Roberto Bifulco, Giuseppe Siracusano

List of Reviewers

Participant	Short Name	Contributor
i2CAT Foundation	i2CAT	Apostolos Papageorgiou
Virtual Open System	VOSYS	Teodora Sechkova
Nextworks	NXW	Gino Carrozzo

Change History

Version	Date	Partners	Description/Comments
v0.1	01-09-2019	NEC	ToC
v0.2	20-09-2019	NEC	Contributions to the various sections
V0.9	25-09-2019	I2CAT, VOSYS, NXW	Reviews
v1.0	30-09-2019	NEC	Final version for submission

DISCLAIMER OF WARRANTIES

This document has been prepared by 5GCITY project partners as an account of work carried out within the framework of the contract no 761508.

Neither Project Coordinator, nor any signatory party of 5GCITY Project Consortium Agreement, nor any person acting on behalf of any of them:

- makes any warranty or representation whatsoever, express or implied,
  - with respect to the use of any information, apparatus, method, process, or similar item disclosed in this document, including merchantability and fitness for a particular purpose, or
  - that such use does not infringe on or interfere with privately owned rights, including any party's intellectual property, or
- that this document is suitable to any particular user's circumstance; or
- assumes responsibility for any damages or other liability whatsoever (including any consequential damages, even if Project Coordinator or any representative of a signatory party of the 5GCITY Project Consortium Agreement, has been advised of the possibility of such damages) resulting from your selection or use of this document or any information, apparatus, method, process, or similar item disclosed in this document.

5GCITY has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 761508. The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed in the deliverable lies entirely with the author(s).

Table of Contents

Figures

Tables

Executive Summary

New applications increasingly rely on swift responses provided by advanced software systems, which usually employ some form of machine learning techniques. These techniques leverage large amounts of collected data to derive mathematical models of the observed reality, in an automated way. In the last decade the set of machine learning techniques known as Deep Learning provided significant breakthroughs in computer vision and natural language processing tasks. This has enabled an entire new set of applications, including those explored by the 5GCity project in relation to video processing and automatic recognition of targets from continuous video streams (ref. work of 5GCity Use Case 1 on unauthorized waste dumping detection).

Leveraging Deep Learning in a distributed setting, such as a city infrastructure, is particularly challenging. Deep Learning is a technology that requires significant computational resources, and as such it best suites datacenter deployments. This hardly matches the requirement or many time-sensitive applications, for which the round-trip time towards a remote, potentially distant, location incurs unacceptable delays.

In this deliverable we devise techniques and solutions to enable Deep Learning to run in the context of a Smart City infrastructure, and in particular, in the distributed setting provided by the 5GCity architecture. In this setting, we envision three main logical execution points for Deep Learning applications, corresponding to the core NFVI (datacenter), the edge NFVI (metro aggregation cabinet) and to the extended edge NFVI (far edge equipment distributed in the city, e.g. a lamppost). Taking advantage of these three execution points requires solving a number of technical issues, which have to do with the ability to run Deep Learning algorithms on heterogeneous systems with very different resource constraints.

In this report we detail our work to address these issues. We first provide an in-depth analysis of the characteristics of the main Deep Learning algorithm, neural networks. Then we evaluate option to distribute such computation, at the high-level, between different executors. Note that in D4.1[19] we explored the opportunity of parallelize the execution of a ML model between different executors exploiting model parallelism[20]. When a model parallel paradigm is applied, the network is “horizontally sliced” and the computation divided among executors. In this deliverable we present a different paradigm: we evaluate the possibility of “vertically” splitting the network in different chunks, and distribute their execution to heterogeneous executors. The selection of the best suited executors depends on the type of computation required by the network chunks (i.e.: compute bound or memory bound). In such a context, we devise an approach to adapt the split computation tasks to the target executor’s hardware. In this we leverage quantization techniques, which allow us to adapt the processing load also to less powerful executors that may be deployed at the edge. In this context, we focus on hardware that is likely deployed at edge nodes, such as modern network processors and SmartNICs. These devices are being increasingly deployed at edge to face space constraints in such edge infrastructures, and are therefore immediately available as additional computational resources. Finally, we provide a software stack to enable the execution of neural networks on processors that are deployed at the extended edge. We do so by devising optimization techniques that reduce the processing load of a neural network, and designing code synthesis algorithms to quickly adapt neural network described in high-level frameworks, such as PyTorch, to typical edge processors based on ARM architectures.

Given the highly experimental nature of this line of work, we have performed a throughout evaluation of the presented approaches in a lab setting, leaving the integration of the proposed techniques in end-to-end applications to future work.

1. Background on Deep Learning

Neural Networks (NNs) are a tool for machine learning that has recently outperformed other approaches in classification tasks, for instance in computer vision. The widespread application of the technique in different fields is significantly impacting datacenters' workloads, which now dedicate a large amount of computing resources to the task of executing NNs [1].

A NN is a collection of interconnected neurons organized in layers. A neuron takes inputs (called Activations) from neurons of previous layer and sends its output to neurons in the next layer. In general, one can think of a layer as a black box that takes an activations vector and transforms it in an output vector. The number of layers, their types and the number of neurons per layer are the hyper-parameters of a NN that finally impact the ability of the network to perform the task at hand.

NN workload can be divided into two phases. The training phase is used to learn the NN's weights, fitting the NN to the specific task at hand. Once trained, the NN is used for the inference phase. Usually the training is a purely offline task, computationally expensive but performed with relatively low frequency. The inference phase is much lighter, but performed many times with potentially very strict latency constraints, since the NN's prediction may be used as part of an online service. Being computationally expensive, the training phase is generally performed by specialized hardware accelerators, for instance general purpose GPUs. These devices can perform parallel computation on large amounts of data, making computation quicker and cheaper, therefore economically viable. Nonetheless, the use of an accelerator incurs some overheads. First, data needs to move within a compute node, i.e., from the general purpose CPU where it is pre-processed, to the accelerator that is attached to, e.g., the machine's PCIe bus. Second, current accelerators provide higher efficiency when performing parallel computation on larger batches. The additional data movements and the need for batching finally affect the overall end-to-end latency. As a result, it is usually more efficient to use of regular CPUs for the NN inference phase, especially when handling latency sensitive workloads [2].

1.1. Neural network types

Neural networks can be composed by layers of different type, depending on the layers used to form the network, a complete taxonomy of the different NN types is out of the scope of this document. However, a brief description of the most common NN types will be useful to understand the different kind of Deep learning workloads:

- Multi-layer Perceptron[26] (MLP)..
- Convolutional Neural Network[25] (CNN).
- Recurrent Neural Networks [27] (RNN).

While these neural network architectures share many common types of layers, they generally provide better results for different specific tasks. For instance, MLP are a good fit to perform tasks on tabular data, convolutional neural networks are capable of dealing with data that show some spatial correlation, such as images, and recurrent neural networks are generally well-suited to work with time series. It's interesting to notice that despite the growing popularity of CNN and RNN, in terms of inference workload composition,

MLPs are by far the most common NNs, followed by RNN and CNN, constituting 61%, 29% and 5% of the workload, respectively [2].

1.2. Neural network Accelerators

NN accelerators, such as general-purpose GPUs (GPGPUs), are commonly deployed as an additional hardware board attached to a machine's PCIe bus. These devices generally use a Single Instruction Multiple Data (SIMD) paradigm to perform large parallel computations. Thus, they are efficiently used when a given task is executed on a large amount of data, for which batching is typically employed. Therefore, from a system perspective, the use of an accelerator to perform NN inference unfolds as follows. First, a request containing the inference task is received through the machine's NIC and sent over the PCIe bus to the RAM. Second, the CPU performs pre-processing of the request to extract the NN input, and adds it to the current batch. Third, once the batch is composed, it is transferred to the accelerator where the inference is performed. Fourth, the result is copied back to main memory, over the PCIe bus. Finally, the CPU prepares the result for transmission to the network, i.e., placing it in network packets, and transfers it, once more over the PCIe bus, to the NIC. We expect a front-end server will receive such result and include it in a user-facing service. The need for batching and the data movements over the PCIe bus add significant latency to the end-to-end processing of the NN inference. For instance, [3] reports that, in their specific settings, the batch size for a GPGPU has to be reduced from 64 to 16 to meet processing latency constraints. This forces the GPGPU to work at 37% of its peak performance (achieved with a batch of 64 in such case). Furthermore, even if Google designed and deployed an accelerator, i.e., the Tensor Processing Unit(TPU)[18], explicitly targeting inference workloads, in some cases the time to transfer data to/from the TPU can be as much as 71% of the time spent during processing.

2. Heterogeneous Deep Learning Inference – Understanding how to split NN inference workloads

The end of Dennard's scaling is challenging the ability to further scale CPUs' computing power [4]. As a matter of fact, Facebook reports that more efficient single CPU machines are replaced with less efficient but more powerful multi-CPUs machines to match required performance levels, for some current machine learning workloads [2]. The "power wall" is now arguably the defining limit of CPUs performance scaling. In order to by-pass the wall chip manufactures and big companies [5] have started to develop and adopt domain-specific processors (DSPs) and specialized hardware to offload some tasks from the CPU. Edge clouds have limited computation resource and energy but still stringent performance requirements in terms of throughput and latency, thus nowadays is common to have edge cloud micro-servers provided with hardware accelerators such for instance SmartNICs [6]. In this section we will profile the NN inference workload and understand if it is possible to distribute such workload between different executors, in order to exploit the availability of heterogeneous executors.

2.1. Deep Learning workload analysis

In this section we analyse the processing required by NN during inference. We perform both a static analysis derived from the model structure and a runtime analysis of the system's resources used during the processing on CPUs.

Since MLPs can be thought as a special case of CNNs, in the rest of the paper we will focus on CNNs, and in particular on AlexNet[8]. In effect, recent NNs may have a bigger number of layers, however, since additional layers are just repetitions of those presented in[8], the AlexNet model remains a valid example while simplifying exposition given its relative simplicity.

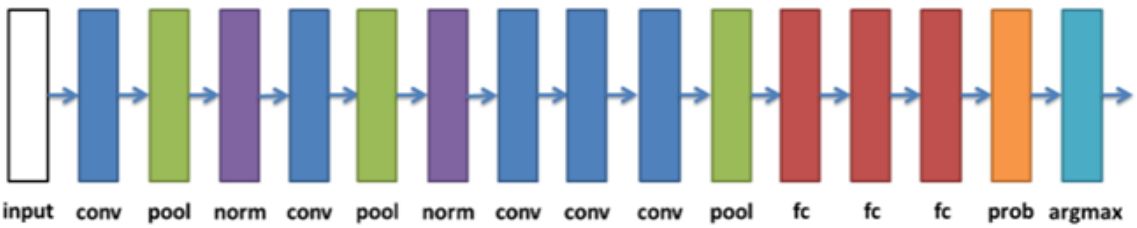

Figure 1: Sequence of layers of AlexNet

2.1.1. Static analysis

A NN layers' number of parameters, and the size of the activations and output vectors, is fixed for a given network model. Figure 2 shows the size of the output vector of each AlexNet's layer (notice that the initial input size is represented by the left-most bar of the histogram). It's interesting to notice that some layers have output vectors bigger than their activations vectors (e.g. conv2 has an input with 69K elements and

produces 186K elements), while for other layers this is reversed (e.g. pool1 layer sensibly reduces the number of output elements from 290K to 69K). Also, it's worth noticing that fc layers have far smaller activations and outputs vectors than other layers, never bigger than 9K. Interestingly, Figure 2 shows that the number of parameters follows a different distribution: conv layers have at most 885K parameters (conv3), while fc have far more parameters, topping to more than 37M (fc6).

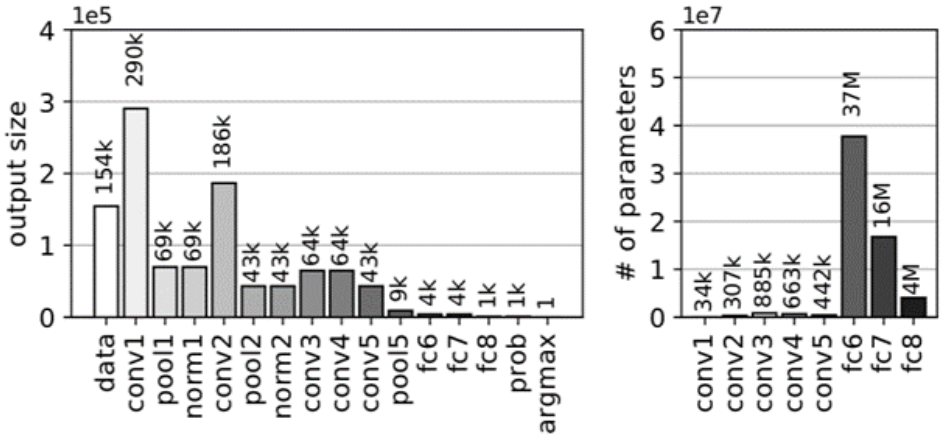

Figure 2: Per-layer output size and parameters:

2.1.2. Runtime analysis

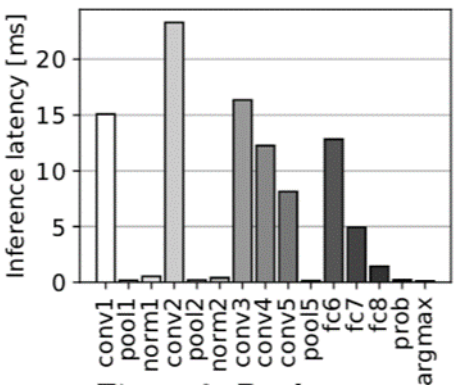

Figure 3: Per-layer inference latency in Alexnet

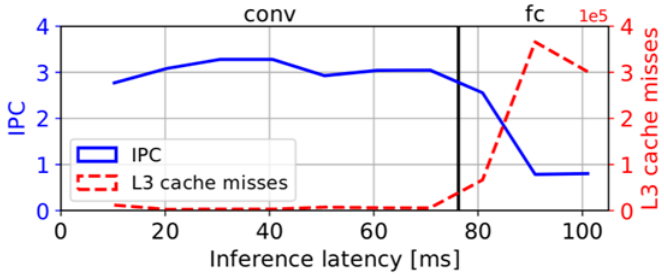

Figure 4: IPC achieved during the NN execution

We run AlexNet on a testbed composed of a dual-socket (NUMA) machine, running Linux kernel 4.10, with two CPUs Intel Xeon E5-2650 (8 cores @2.4GHz), hyperthreading disabled, and 16GB of RAM per socket. Each CPU has 32KB, for both L1 data and instruction caches, and 256KB of L2 cache per core. 20MB of L3 cache are shared by all the CPU's cores. We use perf[1] to collect performance counters, with a 10ms polling interval and avoiding hardware counters multiplexing. AlexNet is implemented using the Caffe framework optimized for Intel processors [7] to take advantage e.g., of the CPU's vector processing features. We instrument the system to collect both the total and per-layer inference latency. Furthermore, we collect hardware counters to measure the Instructions Per Cycle rate (IPC), the number of stalled cycles and the L1 (data), L2, L3 cache misses during execution. All the measurements are executed using a single isolated core on a dedicated CPU (i.e., all the other cores of the same CPU are idle and therefore the L3 cache is

[1] http://man7.org/linux/man-pages/man1/perf.1.html

completely dedicated to the core running AlexNet).**¡Error! No se encuentra el origen de la referencia.** Figure 3 shows the time required to execute each layer of AlexNet when processing a single activation vector. Among the layers, conv2, conv3 and conv1 take the longer to execute, followed by fc6 and then the others. Looking at the hardware counters, we derive the IPC achieved during the NN execution (solid line of Figure 4). We discover the execution of the layers up to pool5 yields an IPC of 3.2, meaning that the CPU's pipeline is filled and that the processing is computation-bound. Instead, the execution of the fc layers achieves an IPC below 1. This lower value is due to a high number of stalled cycles, caused by cache misses in all the cache levels. I.e., the system has to wait for data reads from the RAM. The dashed line of Figure 4 shows the L3 cache misses increment for fc layers.

This performance is quickly explained by the type of computation performed by an fc layer, which does little reuse of the weights' values. This, together with the large number of such layers' parameters, yields an inefficient use of the CPU's caches. For reference, just fc6's parameters need 151MB of memory, considering that each parameter is represented on 32b, i.e., 7 times the space available in the CPU's caches.

2.2. Neural Network Split

The runtime analysis of AlexNet shows that CPUs are efficient executors for conv, pool and norm layers, and far less efficient ones when it comes to the processing of fc layers. That is, the CPU's pipeline is stalled for a large fraction of time during the processing of fc layers, which effectively wastes otherwise useful computation power. Batching improves the situation, but it is a non-viable approach for latency sensitive online workloads.

Taking into account that NNs composed only of fc layers constitute the vast majority of NN workloads, it is clear that developing a solution to improve the execution of these layers can provide important benefits. That is, a suitable executor for fc layers could on one side lower the inference processing latency, and on the other side free CPUs resources that could be better used for other workloads.[2]

It is worth to evaluate and quantify the additional overheads that may be introduced by a split of the NN processing between two different executors.

This will help in identifying the issues in using current accelerators. In particular, we first verify if splitting the execution among two executors has an impact on the processing efficiency, e.g., due to potential cold cache effects or for the inability to leverage some data pre-fetching. Then, we quantify the cost introduced by additional data movements between executors.

2.2.1. Split processing overhead

To evaluate the impact on the processing efficiency, we select two homogeneous executors, but with independent memories and caches, i.e., the two CPUs of our NUMA machine. Being able to exchange data using the fast QPI inter-connection between their sockets, using the CPUs minimizes the overhead of data movements. Moreover, since we use homogeneous executors, we can directly compare the measurements for a split execution to the measurements of a non-split one.

To instrument our testbed, we modify our NN implementation to spawn two processes, each one executing a portion of the NN. The first process performs its layers processing up to the split point, then transfers the intermediate result to the second process that continues the processing till the end of the NN. Both

[2] In effect, stalls may be mitigated by performing a context switch while waiting for the data to be loaded, however this would require sharing the CPU with other tasks. This is likely to increase the processing latency, and could be therefore not viable for latency sensitive workloads. Regardless, since the L3 cache is anyway shared, sharing the CPU with other tasks may in fact be counterproductive

processes load the network model in their respective NUMA node's local RAMs. QPI bars of Figure 5 show that there is an overhead in splitting the NN execution, which can cause up to 8% higher inference time when compared to the non-split execution case, and almost no increases in some cases. The overhead is higher when the splitting happens after a layer with a large output vector.

The test results suggest that the overhead is dominated by the need to wait for the loading of the input vector in the cache of the second CPU. This explanation is easily verified noticing that the overhead of splitting the execution before layers with smaller inputs is always within a modest 1.3%.

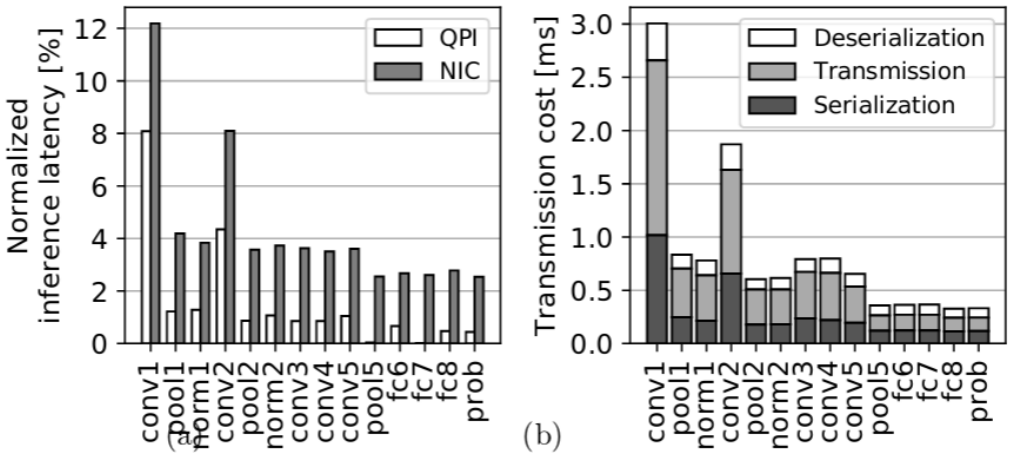

Figure 5: Normalized inference latency and communication overhead for different split points

2.2.2. Communication overhead

In this second test, we split again the execution of the processing between the two CPUs of our NUMA machine, however, this time we force the communication between them to happen through the NIC. That is, we connect the NIC's ports together with a cable in a loop, and modify our NN implementation to transfer the intermediate data (at the split point) using a TCP connection. To avoid the overheads of connection establishment and TCP slow-start, the connection is pre-opened and configured with a very large initial congestion window. The two end-points of the connection live in different Linux network namespaces, ensuring that the communication actually happens through the machine's NIC.

As expected, the end-to-end processing latency grows. NIC bars of Figure 5.(a) show that the latency's increase ranges between 2-12% and follows a pattern similar to the one of QPI bars.

Figure 5.(b) shows the breakdown of the overhead due to communication in three components: (i) data serialization time, required to prepare the intermediate result for transmission; (ii) transfer time, needed to send the data over the TCP socket and to receive it at the other end, e.g., it includes the time to transfer data through the PCIe bus and the NIC's ports; (iii) data deserialization time, required to prepare the received data for feeding it to the second portion of the NN. Of the three components, the transmission time is significantly big only when the split happens after a layer with a large output vector, otherwise it only contributes from tens to hundreds of microseconds. The combination of serialization and deserialization times, instead, always provide about 300us of additional latency, which may be significant on smaller networks. For instance, the three fc layers at the end of the network take about 19ms in total for processing, therefore an additional ms would correspond to about a 1.5% latency increase.

2.2.3. Comments

The tests presented in this section confirm and clarify the issues in using current accelerators for performing NN inference. On the one side, the overhead of splitting a NN execution can be limited if the split point is carefully selected. This suggests that using an accelerator to perform part of the NN processing is possible. On the other side, the cost of moving data to the accelerator may be significant, in particular for smaller networks, confirming the numbers reported in [3]. However, the overhead of moving data is a big issue when using off-path accelerators, such as GPGPUs or TPUs. If the accelerator is located on path, such extra data movement is not required. That is, if on-path devices could perform NN processing, they would introduce mainly the overhead due to the splitting of the processing. Such overhead, as we have seen, can be very little if the split is carefully done, i.e., taking into account layer input sizes.

3. Heterogeneous Deep Learning Inference – Exploiting on-path executors

Artificial neural networks' fully-connected layers require memory-bound operations on modern processors, which are therefore forced to stall their pipelines while waiting for memory loads. Computation batching improves on the issue, but it is largely inapplicable when dealing with time-sensitive serving workloads, which lowers the overall efficiency of the computing infrastructure.

The memory-bound nature of this MLP workload is evident in Figure 6, which reports the IPC of an Intel CPU when processing a three layers MLP with about 121M parameters. The hardware processing pipeline of that CPU is able to achieve an IPC above 3 when used efficiently, that is, it can perform three operations in a single clock cycle. Conversely, when processing the MLP's FC layers, it provides less than 0.5 IPC, since the CPU is forced to wait for data loads. This is due to the relatively low arithmetic intensity of FC layers processing.

Batching several inputs together, and performing NN inference on them at once, allows for re-using a loaded model's parameters several times, thereby increasing the arithmetic intensity. However, increasing the batch size makes inference latency grow. For example, in Figure 6, changing the batch size from 1 to 128 increases the IPC from 0.5 to 2.7, but it also increases the processing latency by almost 10x, from 40ms to about 400ms. For time-sensitive serving workloads, which may include the inference result as part of a user-facing service, such increase is usually not tolerable [9]. Therefore, the maximum batch size is limited to rather small values.

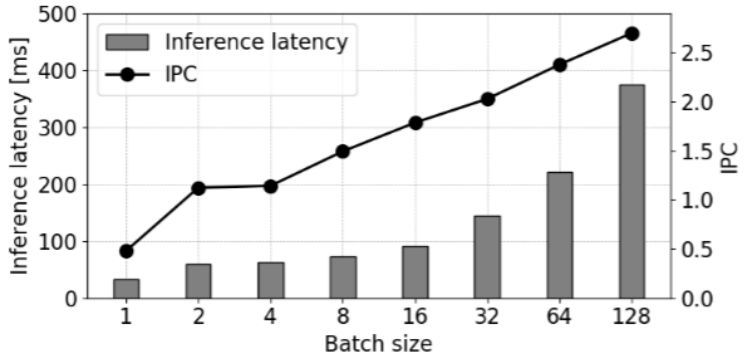

Figure 6: Latency-efficiency trade off

In this section we consider the option of improving a NN serving system's efficiency and throughput using hardware deployed in a machine's networking subsystem and already deployed on the path of the communication, i.e., at the Network Interface Cards (NICs). More specifically, we believe that network packet processing and NN memory-bound inference workloads may have complementary traits, which makes processors developed for networking re-usable, to some extent, for the efficient processing of NN inference. In fact, high-end NICs [10] can handle tens of Gigabits per second (Gbps) of data, they sit already on the data path of an inference request, and are designed to take advantage of the mostly parallel nature of the network traffic processing, which happens on a per-packet basis and nicely fits the per-neuron widely parallel processing model of NNs. Notice that SmartNICs are expected to be widely deployed at MEC nodes.

3.1. Model Quantization

Quantization is the process of reducing the number of bits that's represent a number. Usually 32-bit floating points are used to represent NN weights and activations, however it is possible to reduce the precision of NN inputs and parameters, e.g., using 8bit values, thereby reducing memory requirements, while having little impact on model accuracy. Here, a promising direction are binary NNs [11], which use just one bit for both NN's inputs and activations, while achieving somewhat surprisingly good accuracy results. Table 1 shows it, reporting the accuracy of a binary MLP for an MNIST[21] classification task, and a VGG16 [12] modified to replace the final 3 FC layers with binarized FC layers, for classification on the CIFAR10 dataset [22].

	MLP	Bin-MLP
Dataset	MNIST	
Layers	3 (bin)FC	
Acc.	98.7%	98.4%
	VGG	Bin-VGG
Dataset	CIFAR10	
Layers	13 conv, 3 (bin)FC	
Acc.	94.0%	94.0%

Table 1: Regular vs binarized NN

Interestingly, binarized models have also the effect of simplifying the arithmetic operations required to compute a NN, replacing matrix multiplications with bitwise operations, such as XOR. This enables the use of a machine's hardware component that were originally designed for different tasks. Taking advantage of this observation, we consider the option of improving a NN serving system's efficiency and throughput using hardware deployed in a machine's networking subsystem, i.e., at the Network Interface Cards (NICs).

3.2. Running an Inference server on a SmartNIC

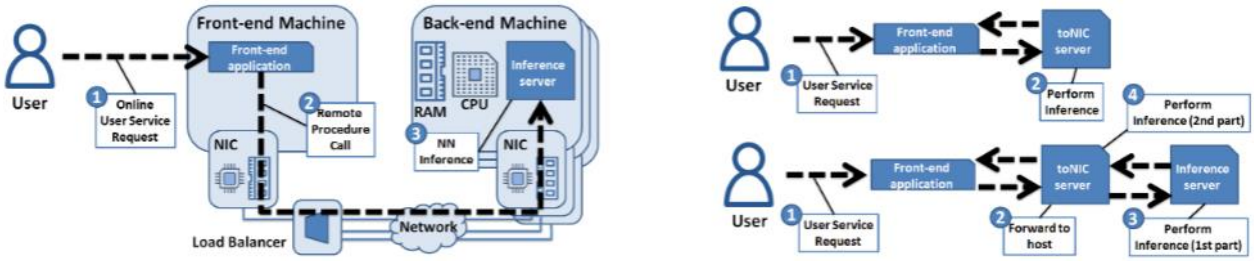

Figure 7: NN system overview and toNIC end to end deployment

This section presents the design and implementation of *toNIC*, an inference server that runs on SmartNICs and can serve inference workloads for binarized NNs composed of FC layers. A high level description of an end-to-end inference system [13][14] is presented Figure 7, the system composed by at least a Front-end machine and a set of Back-end machines used to perform NN inference. The Front-end runs the front-end application, handles users' requests and forwards them to the inference server, of course the communication between such machines happens trough the networking subsystem. The *toNIC* inference server will run inside the Network cards of the back-end machine and will be able to work in two different configurations:

- **Stand-alone mode**, the whole NN inference is performed on the NIC (e.g., for MLPs inference).
- **Co-processor mode**, the NN inference is performed in combination with a traditional NN serving framework (e.g., for convolutional NNs inference). That is, for cases in which FC layers are placed at the end of a NN, like in some convolutional NNs, the compute-intense convolutional layers can be processed by the inference server on the machine, while the memory-intense FC layers are offloaded to *toNIC*.

3.2.1. SmartNIC target

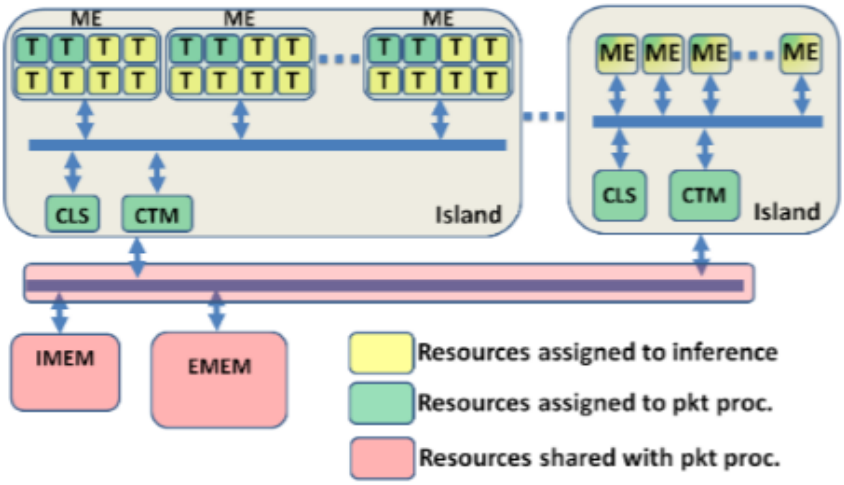

Figure 8: Inference and packet processing threads distribution.

Programmable NICs include a processor, often specialized for the processing of network packets, a variable amount of DRAM, and one or more network ports. The processors can be of different types, including general purpose CPUs [16], specialized network processors [10] and even FPGAs [15]. In this work, we will focus on network processor-based SmartNICs. These can have price and power consumption comparable to traditional NICs. Furthermore, they are generally faster in handling network packets when compared to CPU-based SmartNICs, which makes them an increasingly common choice for several deployments [17]. In particular, we will use Netronome SmartNICs based on the NFP4000 processor The Netronome NFP programmable architecture is shown in Figure 8. Since network traffic is a mainly parallel workload, with packets belonging to independent network flows, NFP4000 devices are optimized to perform parallel computations, with several processing cores (micro-engines, MEs). Each ME has 8 threads, which share local registries that amount for a total of 4KB. MEs are further organized in islands, and each island has two shared SRAM memory areas of 64KB and 256KB, called CLS and CTM, respectively. Finally, the chip provides a memory area shared by all islands, the IMEM, of 4MB SRAM, and a memory subsystem that combines two 3MB SRAMs, used as cache, with larger DRAMs, called EMEMs. MEs can communicate and synchronize with any other ME, irrespective of the location (i.e., same or different islands).

3.2.2. toNIC Design

The main requirement of *toNIC* is to provide its inference function while preserving high performance network communications. In our current design, we clearly separate the two functions, dedicating a configurable number of MEs' threads to the inference execution and the remaining ones to packet forwarding. Figure 9 shows that we distribute the MEs' threads dedicated to inference across different islands. This is somewhat counter-intuitive, as one would expect that a co-location of processing elements working for the same task would yield better performance and more efficient synchronization. However, it should be noted that each island is provided with the fast CLS and CTM memories, which play an important

role during packet processing. In fact, packet processing happens on a nanoseconds time scale, which implies a small time budget allocated to the processing of each packet. Instead, NN inference usually happens in a time budget of hundreds µs. Therefore, having access to fast local SRAMs is required to provide packet forwarding at low latency and high throughput, and assigning all the MEs of an island to inference would immediately make that island's fast memories unavailable to the packet processing task.

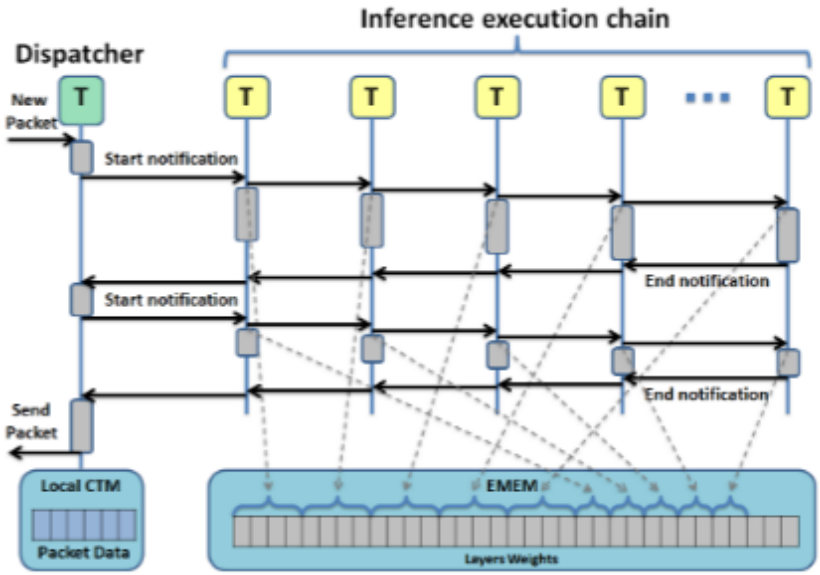

Figure 9: Inference threads notification and execution process.

Note that this design decision has the immediate effect of limiting the usefulness of batching computation for inference in *toNIC*. That is, weights cannot be cached in a local memory, being it dedicated to packet processing.

To combine forwarding and inference tasks, the packet forwarding threads work also as dispatchers for the incoming network packets that contain an inference request, e.g., encoded in an efficient remote-procedure-call (RPC) protocol. More in detail, at NIC's boot, dispatching ME's threads registers themselves for packet reception notifications. When a new network packet is received, the NFP4000 copies the packet content in the CTM memory of a dispatching thread's island, and generates a reception notification. At this point, the dispatching thread can parse the packet and invoke the forwarding function, if the packet is not destined to *toNIC*, or the inference function otherwise. The forwarding function is executed together with other non-programmable subsystems of the NFP4000, while the inference function is executed by the MEs' threads dedicated to inference.

3.2.3. Inference Processing

A dispatching thread works as a coordinator for the execution of an inference, by starting the processing of one NN layer, and waiting for the result before starting the processing of the next layer. Figure 9 depicts this process. The MEs' threads dedicated to inference are organized in a statically defined chain, with each ME knowing its predecessor and successor MEs' threads at boot time. To start processing a layer, the dispatching thread notifies the first thread in the chain with a start notification, which is then propagated throughout the chain.

After receiving the start notification, and sending it to the next thread in the chain, a thread performs the computation of a subset of the current layer's neurons, with the actual number depending on the number of neurons in the layer and the level of configured execution parallelism. Each of the thread is an executor

from the perspective of *toNIC*, so the configured number of executors (threads) defines the level of parallelism.

For each of the neurons, an executor performs an XOR of the input with the weights vector, a population count operation and a comparison. The NN model input is stored in the packet data, located in the dispatcher thread's local CTM, which can be accessed also from other islands although paying the extra clock cycles required to do a cross-island read. The model's weights are stored in the DRAM-backed EMEM, in a contiguous memory space, which allows an executor to directly point to the weights of a set of neurons given its position in the execution chain. At the end of a layer computation, each executor writes its final result to the global IMEM memory, from which the dispatching thread can read it.

The last executor in the chain sends an end notification to its predecessor after writing its portion of the result to IMEM. The notification is propagated backward through the chain as all executors conclude their computations and write their results, until it is received by the dispatching thread. Here, either the computation for a new layer is started or the result is encapsulated in a network packet. The packet is then finally handled by the forwarding function.

4. Efficient Deep Learning platform for the edge

Having presented a first solution to enable the split execution of Neural Networks at e.g., edge nodes, we now provide a solution to run neural network at the extended edge, e.g., on devices deployed widely throughout the city. For this setting, we need to provide on the one side optimized neural networks that can more efficiently leverage the scarce resources of edge nodes, and on the other side a compatibility layer that allows such computations to take advantage of the heterogeneous hardware usually deployed at the edge.

In this context, it is worth to notice that the quick evolution of Deep Learning has been propelled by the availability of a number of software frameworks that simplify the development of neural network algorithms. Currently, a number of alternatives Deep learning frameworks have emerged, with PyTorch, TensorFlow, MXNet, CNTK being among the most popular deep learning frameworks.

Conceptually, these frameworks provide a ready-to-use set of neural network building blocks, and a high-level API to implement the neural network layers. Writing a new neural network is as simple as sticking together a few lines of code using the python programming language. The frameworks will then take care of all the operations required to enable the training of the neural network and its execution.

Under the hood, each of these frameworks maps the high-level API to low-level optimized hardware-specific computation libraries. For example, the cuDNN library is used with NVIDIA GPUs that use the CUDA computation model. Given the high-level API, the user of the framework can ignore the low-level hardware details, focusing on the neural network algorithm design. This also provides hardware agnostic implementation of the neural network, since the hardware executor, e.g., a CPU or a GPU, is managed by the framework transparently.

However, the high-level API abstractions may also affect the overall efficiency of the implemented low-level computations. For instance, the high-level APIs generally expose neural network layers as atomic components of a neural network, requiring their execution to happen serially on the underlying hardware. In some cases, the execution of different layers could be instead merged together from a computational perspective, optimizing the use of the underlying hardware, e.g., avoiding the data transfer that usually happens when starting and finishing the computation of a layer.

4.1. SOL

SOL[3] is a deep learning platform developed by NEC that aims portability, extendibility, usability and efficiency. Our framework seamlessly integrates into popular frameworks, rather than introducing “yet another API”. As such, in contrast to other techniques, it does not replace any of the original functionality of the original deep learning frameworks but complements them.

Users continue writing neural networks using their favourite frameworks, e.g. TensorFlow or PyTorch, while our platform seamlessly analyses the neural network descriptions to provide additional hardware compatibility and improve performance.

[3] http://sysml.neclab.eu/projects/sol/

SOL reads the description of a neural network defined in the framework's syntax, analyses its structure, and automatically derives a set of optimized computations for the specific neural network the user has described. Then, in addition to the low-level computation functions available in libraries such as NVIDIA cuDNN or Intel MKL-DNN, our platform also generates any additional low-level computation function that may be required by the optimized neural network, for a target hardware executor. That is, using automated code generation techniques, we are able to create on the fly a computation library that is optimized specifically for the user's neural network. The generated code is compiled in implementations that are fully functionally equivalent for the user, which can perform the exact same set of computations on the data, but more efficiently.

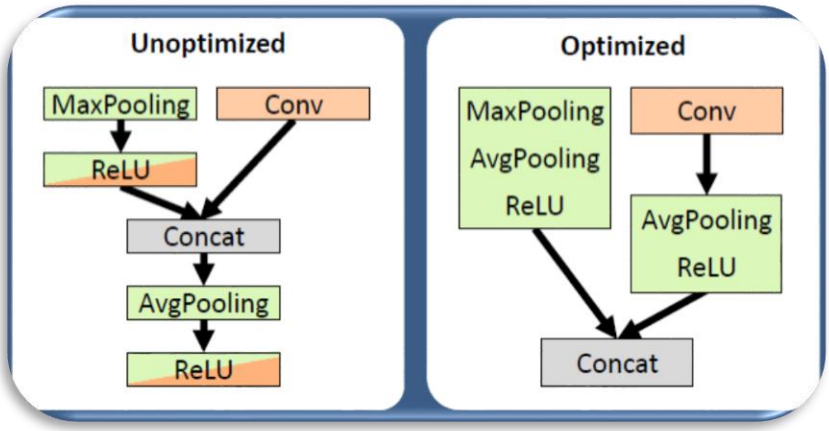

Figure 10: High level view of SOL optimizations.

4.1.1. Optimization techniques

Deep Learning frameworks, like TensorFlow and PyTorch, execute their neural networks on a layer-by-layer basis, which can result in an inefficient use of the hardware executor's memory hierarchy and thereby in long execution times. Existing deep learning frameworks parallelize the computation graph in a breadth-first manner, finishing all computations at one level of computation before starting the next level's computations. We refer to this as breadth-first parallelism because the operations at one NN layer are executed in parallel before the computation proceeds to lower NN levels.

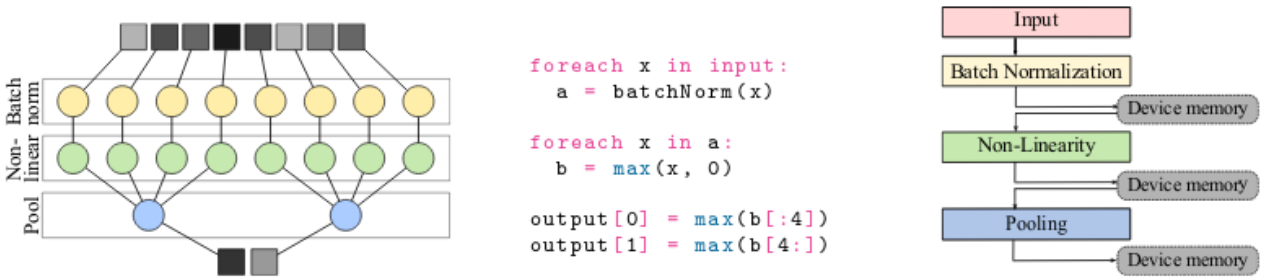

Figure 11: Breadth-first parallelism

In Figure 11 we show an example of breadth-first parallelism, the computation graph (left) for the neural network layers (right) and the corresponding code snippet (middle). The standard method for processing the layers is in a breath-first manner, with every operation of level *i* in the computation graph executed in parallel before operations at level *i+1*. This is indicated by the white boxes (left) surrounding each level of the computation graph. The problem with this approach is that each layer generates relatively large temporary data that cannot fit in GPU's/CPU's fairly small caches.

SOL is able to detect and execute more complex independent computation paths in parallel. While this does not reduce the overall number of operations, it often leads to situations where the data accessed by these independent paths fit into the caches and registers of the hardware, increasing performance. We refer to the parallel processing of independent paths in the computation graph as depth first parallelism.

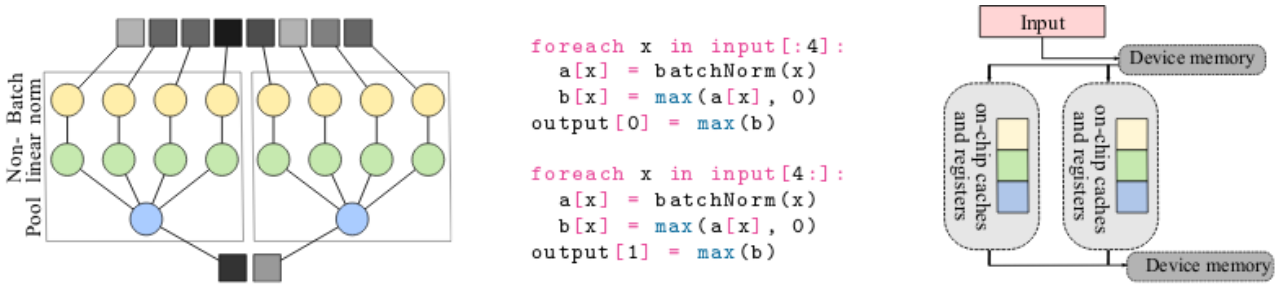

Figure 12: Depth-first parallelism

In Figure 12 we show an example of depth-first parallelism, the computation graph (left) for the neural network layers (right) and the corresponding code snippet (middle). SOL can detect independent paths in the computation graphs and aggregate these paths into independent processing blocks. The white boxes (left) indicate the parts of the computation graph that SOL chose to parallelize. The intermediate data generated within these blocks fit into the hardware cache.

4.1.2. Software compatibility

The speed at which deep learning architectures are developed, tested, and open-sourced is staggering. For instance, almost every other week a pre-trained model for natural language processing is published (such as ELMO, BERT, XLNet) and shown to be superior to previous ones on established benchmarks. In many cases, these models are written in one specific deep learning framework, optimized for efficiency, and pre-trained on very large datasets. To allow businesses to utilize these models is straight-forward if the deep learning framework matches that of the existing development environment. In many cases, however, there is a mismatch and it is time consuming and expensive to port new architectures and layers to another framework. It is time consuming because the implementations are highly optimized and, therefore, non-trivial to port to other deep learning frameworks. It is expensive because pre-training a model often takes several days on specialized hardware.

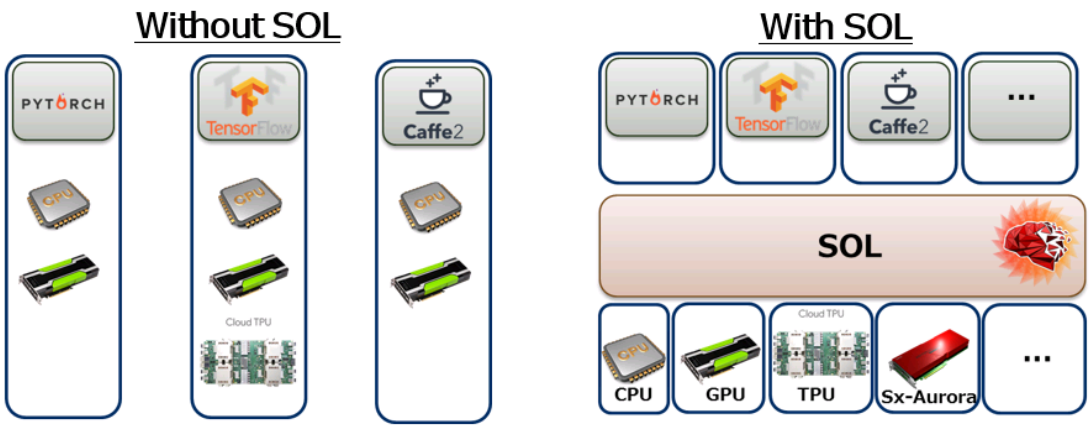

Figure 13: SOL high level architecture.

SOL, working as a middle layer between specialized hardware and deep learning frameworks (Figure 13), allows one to design and experiment with deep learning models with cross-framework layers. For instance, we can use SOL in Tensorflow to include PyTorch layers and vice versa. For example, TensorFlow does not

support the ConstantPadding layer, which is supported in PyTorch instead. Our framework enables the use of these layers across frameworks.

4.1.3. Neural Network Deployment

When a neural network model has been trained, it is supposed to be integrated into an application. Since models are developed using a Deep Learning framework, running the developed neural network requires shipping the entire framework together with the application. This is often too inefficient, in particular for some deployments where performance is critical. As such, often a developed neural network goes through expensive engineering efforts to port it into a stand-alone implementation that can be run independently from the original framework. To help the deployment model, some frameworks provide leaner libraries that can be included in applications. For instance, PyTorch provides LibTorch, which is a C++ interface capable of running previously trained PyTorch models. However, this package alone is over 700MB, which would need to be distributed with the target application. Toolkits such as NVIDIA's TensorRT require the neural network to be explicitly converted into their own format, facing again significant engineering efforts and compatibility issues, such as missing layer implementations.

SOL supports the deployment of the neural network models for any supported hardware. Using a deployment function "dlp_model.deploy(target=dlp.target.linux_shared, device=dlp.device.ve, name='predict_model')" is all that is needed to generate a shared Linux library that provides the function "predict_model(input, output)" that runs the network on the target device. The deployment function relies on the same optimization engine and architecture of SOL The generated libraries only contain the neural network execution functions, parameters and a minimal set of helper functions, which significantly reduces the size of these libraries and simplifies their integration in the applications that need them.

5. Proof of concept evaluation – *toNIC*

We evaluate *toNIC* components and the system as a whole, measuring key performance indicators such as throughput, latency and power consumption. We compare our prototype with a serving system for binary layers optimized for the Haswell CPU - bnn-exec - which we developed specifically for this evaluation, to leverage Haswell's AVX instructions[4]. *toNIC* runs on the NFP, while bnn-exec runs on the CPU). Both execute a single FC layer (including activations) with 4096 binary inputs and a variable number of neurons, from 2k to 16k. When doing latency measurements, we use a single core of the Haswell, while we consider all 4 cores for throughput benchmarks. Instead, *toNIC* always runs using 256 executors (6 threads from each of 42 MEs, plus 4 threads from another ME).

5.1. Latency

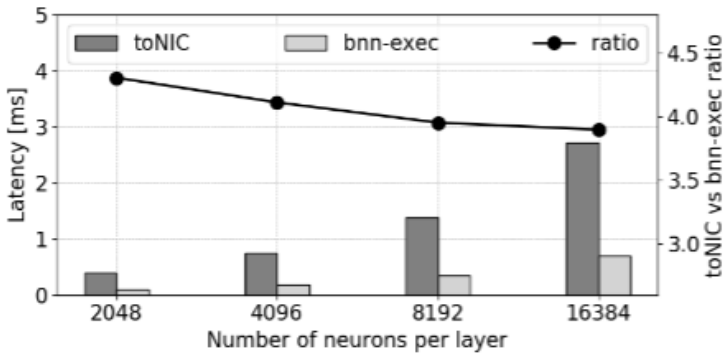

Figure 14: The processing latency of an FC layer.

Our first test measures the per-inference latency, when processing a single FC layer (Figure 14). For layers between 2k and 16k neurons (8M to 67M weights), *toNIC* achieves a processing latency which is only 4 times higher than bnn-exec's one, varying between 400 µs and 2700 µs. Considering that the Haswell CPU has a clock frequency more than 4 times higher than NFP's 800MHz, i.e., each operation is effectively executed in less than a fourth of the time, this shows that the NFP is slightly more efficient than the Haswell in performing the FC layer operations. Even considering the larger FC layer with 67M weights, this processing latency should be acceptable for most applications.

5.2. Throughput

toNIC cannot perform batching, since it has to save hardware resources for packet forwarding tasks. In contrast, bnn-exec can batch executions, with the batch size being limited by the maximum allowed per-inference latency. We set this limit arbitrarily to 7ms, referring to [3]. This latency constraint allows bnn-

[4] All tests have been executed on a server equipped with an Intel Haswell E5-1630 v3 CPU and a single Netronome Agilio CX SmartNIC, with an NFP4000 processors and two 40Gbps ports. The Haswell is clocked at 3.7GHz, while the NFP at 800MHz

exec to run with a batch size of 64, 32, 16 and 8 for the 2k, 4k, 8k and 16k neurons layers, respectively. Figure 15 reports the results in terms of FC layers per second (FC/s). For bnn-exec we plot the result obtained for each FC size with the abovementioned batch sizes (and running on the CPU's 4 cores). *toNIC*, despite unable to perform batching, and using only a subset of the NFP4000 resources, can still provide 4-5% of the bnn-exec throughput.

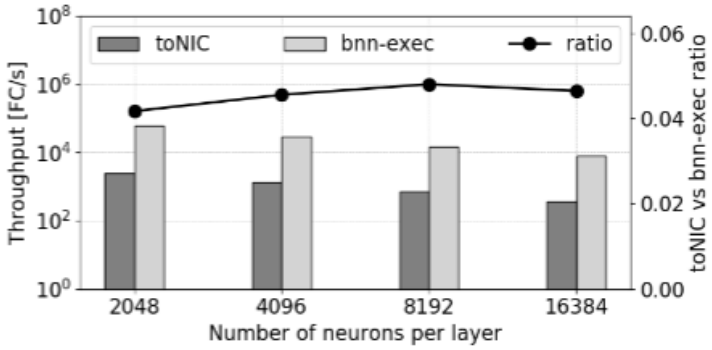

Figure 15: Throughput for different layer size.

5.3. Forwarding performance

While running the above throughput test, we load the NFP4000 with 80Gbps of network traffic to be forwarded (line rate of the NIC). For all packet sizes of 512B and bigger, the SmartNIC could forward traffic at line rate, i.e., both the packet forwarding function and the inference function could provide their maximum throughput while working in parallel.

5.4. Power measurements

	POWER (VS IDLE)	FC/S	FC/W VS IDLE
IDLE	69.4W	-	-
BNN-EXEC	145.9W (+75.5)	29520	386
TONIC	70.2W (+0.8)	1344	1680

Table 2: Power measurements when computing FC layers on NFP and CPU

The previous results tell relatively little about the actual performance of the evaluated systems, which is better captured by a metric related to power efficiency. In particular, we use the FC layers per Watt (FC/W) as an indicator of power efficiency, and as a proxy for the cost of running the system. The power consumption of our test machine when completely idle is 69.4W. For measuring the FC/W rate, we serially execute FC layers with 4k neurons (16M weights, 2MB in memory) on each processor, separately. We select this workload being a middle-ground between the best and worst performance of *toNIC* compared to bnn-exec (cf. Figure 15). Furthermore, this allows both processors to minimize loads from DRAMs, since both of them have large enough SRAM caches to hold the 2MB of weights (EMEM's cache in the case of NFP, and L3 for Haswell). *toNIC* achieves a throughput of about 1.35k FC/s, with the overall machine power consumption rising to 70.2W, i.e., it takes 0.8W additional W of power to compute 1.35k FC/s. This yields a very promising 1.7k FC/W relative to the idle power consumption. When the Haswell CPU is used for executing FC layers, instead, the throughput is 29.5k FC/s, but the power consumption raises to 145.9W,

with a performance of just 386 FC/W relative to the idle power consumption. That is, the NFP4000 yields a4.3x better performance/power ratio (cf. Table 1).

To put this in perspective, we need to take into account the overall power consumption of the machine, and factor in the relative contribution given by *toNIC*. I.e., *toNIC* provides a 5% maximum throughput increment in terms of FC/s to the machine, in exchange of a 0.5% increase in power consumption, thereby increasing the overall machine FC/W by about 3.9%.

5.5. **End-to-end test**

In a final test we measure the end-to-end processing latency for a full inference task, using the MLP example of Table 1. We measure the latency from a second server connected back-to-back with our machine running *toNIC* (bnn-exec), measuring the time difference between an inference response and its request as seen by the second server. The latency with *toNIC* is 1.05ms, while with bnn-exec it is 0,83ms, when both devices have a cold cache. Here, it is interesting to notice that being *toNIC* running in the NIC, of the 1.05ms only 35µs are spent in network packet's reception and transmission, while for bnn-exec it takes 190 µs only to handle the network traffic. Notice that these values include the overhead introduced by the measurement server. The additional latency of bnn-exec is due to the need to traverse the NIC, the PCIe bus and the host's machine network stack before bnn-exec can actually receive the input to perform inference.

6. Proof of concept evaluation – SOL

We evaluate the acceleration that SOL provides when executing commonly used neural networks using the PyTorch framework. NN are models are coded using the PyTorch APIs and the executed respectively using the native PyTorch framework and using the SOL framework.

Conceptually, PyTorch and all the other frameworks provide a ready-to-use set of neural network building blocks, and a high-level API to implement the neural network layers. Writing a new neural network is as simple as sticking together a few lines of code using the python programming language. The following code snippet summarize a typical python script used to load and execute a NN.

```
import framework

model = framework.ModelZoo.init("MyNeuralNetwork", …)
input = (…load input…)
result = model(input)
```

The frameworks will then take care of all the operations required to enable the training of the neural network and its execution.

In order to use the SOL framework while keeping the original framework APIs a user needs only to import SOL, and add one line of code.

```
import framework
import sol.framework as sol

model = framework.ModelZoo.init("MyNeuralNetwork", …)
model = sol.optimize(model)
input = (…load input…)
result = model(input)
```

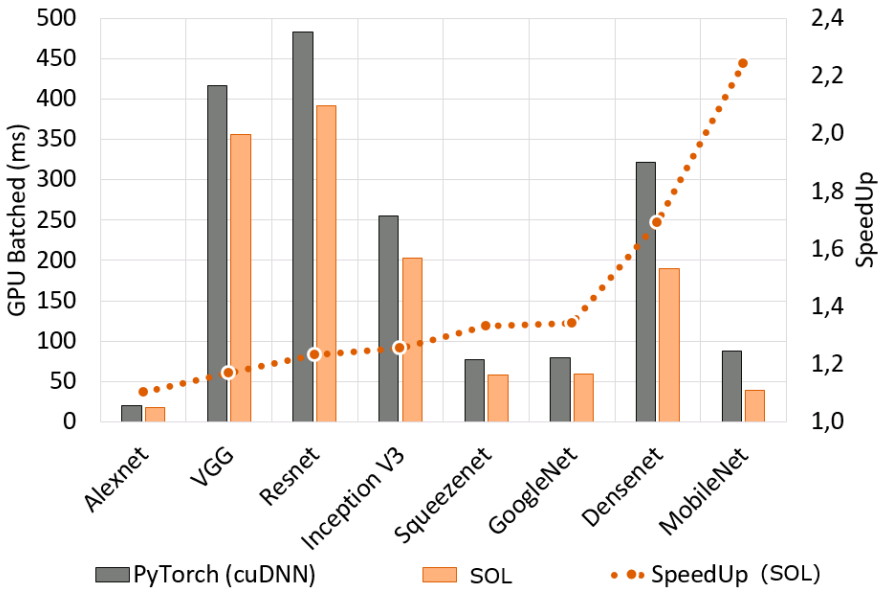

Figure 16: GPU inference time (batch size 128)

Figure 16 shows the total execution time when running the inference on a GPU with a batch size of 128, the networks have significantly varying execution times ranging from very short (AlexNet) to quite long (Densenet-161 and Resnet-152), SOL provides a speed up in all cases, with the most pronounced improvements for Densenet[24] and Mobilenet[23].

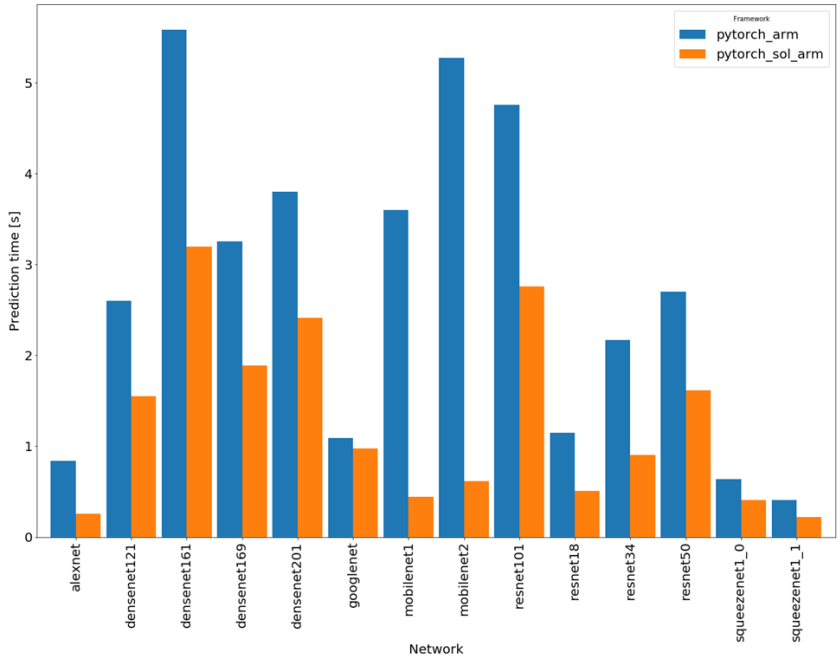

Figure 17: ARM CPU inference (batch size 1)

Figure 17 shows the total execution time when running the inference on ARM CPU with a batch size of 1, this experiment is particularly interesting because the execution target. In facts, ARM CPUS are the de facto standard for edge devices, form the results we ca see that the speed-up relative to standard PyTorch is even more evident. Note that the execution of Mobilenet, an NN explicitly designed for embedded devices, has the biggest speed-up.

7. Conclusion

In this report, we presented an extended analysis of Deep Learning workloads, devising techniques to run neural networks in split mode and on heterogeneous devices. Our work enables the execution of neural networks on top of distributed infrastructure comprising several different executors. We show that commodity SmartNICs and edge executors based on ARM processors can effectively run neural networks. We also describe the design of a software stack that enables the automatic optimization and creation of software adaptation of neural networks, enabling their execution at the edge.

The presented solutions are enabling factors for orchestration systems that can take advantage of such techniques to allocate computing tasks to a larger set of nodes.

In the context of 5G CITY both toNIC and SOL can be applied to improve deep learning performance, and enable the envisioned edge use cases. For instance, one possible application in the context of the illegal waste dumping detection use case is the execution of neural networks directly on the deployed cameras. This would enable the system at saving costly data transfers, or eliminate the need for deploying more powerful servers in proximity of the cameras. In particular, it is possible to use SOL to deploy simpler neural networks that detect activity, and only then trigger the execution of a more complex neural network that detects the specific illegal waste dumping activity. Likewise, toNIC could be employed to further scale the edge datacenters performance, and increase the amount of cameras/network flows that can be handled by a single edge datacenter.

7.1. References

[1] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and *tonic*: Dnn as a service and its implications for future warehouse scale computers. In ACM SIGARCH Computer Architecture News , volume 43, pages 27–40. ACM, 2015.

[2] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 620–629, Feb 2018.

[3] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, pages 1–12. ACM, 2017.

[4] Nikos Hardavellas. The rise and fall of dark silicon. USENIX ;login: , 37:7–17, 2012

[5] Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., ... & Lo, D. (2016, October). A cloud-scale acceleration architecture. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture (p. 7). IEEE Press.

[6] https://www.netronome.com/press-releases/packet-and-netronome-innovate-smart-networking-focused-edge-compute-hardware/).

[7] Vadim Karpusenko, Andres Rodriguez, Jacek Czaja, and Mariusz Moczala. Caffe* Optimized for Intel Architecture: Applying Modern Code Techniques. Technical report, Intel, 08 2016.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[9] Ankit Singla, Balakrishnan Chandrasekaran, P Godfrey, and Bruce Maggs. The internet at the speed of light. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks , page 1. ACM, 2014.

[10] Netronome. Netronome AgilioTM CX 2x40GbE intelligent server adapter, 2018. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.

[11] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR , abs/1602.02830, 2016

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. CoRR , abs/1409.1556, 2014.

[13] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. arXiv preprint arXiv:1712.06139 , 2017.

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In NSDI, pages 613–627, 2017.

[15] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. IEEE Micro , 34(5), 2014.

[16] Mellanox. Mellanox, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[17] OpenComputeProject. SPEC, MEZZ, OCS, Netronome NIC retrieved sept. 2018, 2018.

[18] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[19] 5G City D4.1 Orchestrator design, service programming, and ML models

[20]Jia, Zhihao, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks." arXiv preprint arXiv:1807.05358 (2018).

[21] http://yann.lecun.com/exdb/mnist/
[22] Krizhevsky, Alex, and Geoffrey Hinton. Learning multiple layers of features from tiny images. Vol. 1. No. 4. Technical report, University of Toronto, 2009.
[23] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).
[24] Iandola, Forrest, et al. "Densenet: Implementing efficient convnet descriptor pyramids." arXiv preprint arXiv:1404.1869 (2014).
[25] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proceedings of the IEEE. 1998 Nov 11;86(11):2278-324.
[26] Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Cognitive modeling. 1988 Oct;5(3):1.
[27] Hochreiter S, Schmidhuber J. Long short-term memory. Neural computation. 1997 Nov 15;9(8):1735-80.

Abbreviations

7.2. Abbreviations

CLS	Cluster Local Storage
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CTM	Cluster Threaded Memory
DSP	Domain Specific Processor
EMEM	External Memory
GPU	Graphic Processing Unit
IMEM	Island Memory
IPC	Instruction Per Clock
ME	Micro Engine
MLP	Multi-Layer Perceptron
NFP	Network Flow Processor
NFVI	Network Function Virtualization Infrastructure
NIC	Network Interface Card
NN	Neural Network
NUMA	Non uniform Memory access
QPI	Quick Path Interconnect
RNN	Recurrent Neural Network
TPU	Tensor Processing Unit

<END OF DOCUMENT>