Efficient Neural Computation on Network Processors for IoT Protocol Classification

Vibha Pant*, Roberto Passerone†, Michele Welponer†, Luca Rizzon‡ and Roberto Lavagnolo‡
*Dept. of Computer Science and Engineering, Amrita School of Engineering, Bengaluru,
Amrita Vishwa Vidyapeetham, Amrita University, India
†Department of Information Engineering and Computer Science, University of Trento, Trento, Italy 38123
‡Microtel Innovation Srl, via armentera 8, Borgo Valsugana, Trento, Italy 38051

***Abstract*—The Internet of Things (IoT) brings forth pressing requirements on the service providers in terms of service differentiation, which plays an important role in pricing policies as well as network load balancing. In this paper, we consider differentiation of application level protocols for IoT from general application protocols through flow classification. We implement a neural network classifier that can run at wire speed reaching 100 Gbps on a network processor. In particular, we study approximations which allow us to efficiently compute the neural network output, while complying with the network processor limitations, which does not provide multiplication or other complex mathematical operations. The results show that the implementation is efficient and that the classification error is negligible.**

I. Introduction and Related Work

Traffic and packet classification is an important functionality at the basis of network management activities such as resource planning, quality of service (QoS) provisioning, load balancing and lawful intrusion detection [1]. In particular, the increasing adoption of IoT devices raises specific requirements on network operators, which must be able to distinguish IoT traffic to determine the correct QoS, pricing and provide differentiated services for applications and network monitoring [2]. Packet classification must therefore be performed at wire speed, so that communication flows can be routed to the appropriate processing device in a balanced manner according to their class of service. In this paper, we look at statistical analysis and classification using neural networks implemented on network processor (NP) architectures, which are commonly used when packets must be processed and routed at wire speed, owing to their dedicated hardware for buffering, table search and update, and forwarding [3]. On the other hand, NPs are not designed for complex computation, as their simplified instruction set does not provide multiplication or higher math operations. Our main contribution is a set of approximations that make use of only additions and subtractions, which can be implemented efficiently on NPs and significantly improve performance. Our evaluation shows that the approximations do not affect the classification accuracy appreciably.

Several methods can be used to perform packet and protocol classification. The simplest and most efficient classification technique is port matching, where the destination and source port numbers or IP addresses are matched to a set of well known values using pre-defined rules [4], [5]. This is frequently used in intrusion detection systems on the server side. Port spoofing or camouflaging however make this unreliable and has rendered it obsolete [6]. Packet Payload analysis or Deep Packet Inspection (DPI) achieve high classification accuracy, however in many scenarios the payload is not accessible due to encryption or legal privacy restrictions [7].

In this paper, we opt for Deep Flow Inspection (DFI) where we look at traffic as a flow and which does not require the analysis of the payload. Flow statistics such as flow length, packet size distribution, session start and end time can be used in this classification approach [8], [6]. Flow inspection can be treated as a pattern recognition problem where we can apply machine learning algorithms [9], [10]. We use artificial neural networks, which have shown great performance in classification and clustering of large amounts of data received from sensors [11]. Shen et al. use several statistical properties related to packet size as input features to distinguish P2P traffic [10]. We follow in particular the approach proposed by Trussell et al., who use packet size distribution to classify different application protocols [12]. The authors pre-process the data fed into the neural network and use bins in a histogram to hold the distribution of packet sizes for each application. We use a similar approach to pre-process the IoT data. Unlike previous work, we employ a network processor and take advantage of its inherent parallel and multi-tasking capabilities to support computational intensive classification at wire speed.

We first give an overview of the classification methodology in Section II. We then discuss the approximations required to implement the algorithm on a network processor in Section III. Experimental results are presented in Section IV.

II. IoT Protocol Classification Overview

Our classification strategy uses the packet size distribution of a flow to determine the likely application protocol [12]. In particular, we consider the following three protocols, which are the most used in the IoT: Constrained Application Protocol (CoAP), primarily used for communication in constrained nodes with low power, computation and communication capabilities; Message Queuing and Telemetry Transport (MQTT), which connects embedded devices through a publish, subscribe and broker environment; and Advanced Message Queuing Protocol (AMQP), which provides reliable delivery with primitives such as at most once, at least once and exactly once delivery parameters.

978-1-5090-6447-2/17 $31.00 © 2017 IEEE
DOI 10.1109/NGCAS.2017.55

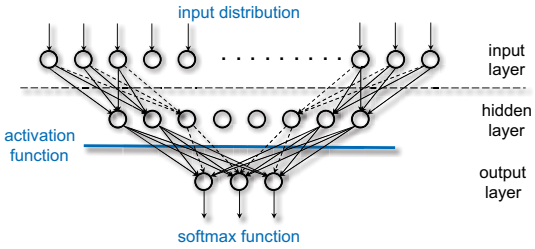

Fig. 1. Neural network diagram

The classification is performed by the three-layer neural network shown in Figure 1. Following [12], the network is composed of $n_x = 50$ input nodes x_i, $n_h = 8$ hidden nodes h_j and $n_y = 3$ output nodes y_k. The packet size distribution is organized in 50 bins, encoding packet sizes in steps of 32 bytes, up to a maximum size of 1600 bytes. The 50 values corresponding to the distribution are fed to the corresponding input layer nodes of the neural network. The input layer is fully connected to the hidden layer through weights w_{ij}^{xh}, while the hidden layer is fully connected to the output layer through weights w_{jk}^{hy}. Before computing the output layer, the values of the hidden layer are processed through an *activation function*. In our case, we use the hyperbolic tangent (tanh). Thus, the output nodes y_k are computed as

$$y_k = \sum_{j=0}^{n_h-1} w_{jk}^{hy} \cdot \tanh(h_j), \quad h_j = \sum_{i=0}^{n_x-1} w_{ij}^{xh} \cdot x_i$$

Each output node corresponds to one of the MQTT, AMQP and CoAP protocols. To correctly classify a protocol, we expect one of the output nodes to be significantly more *active* than the others, i.e., to have a larger value. To check this condition, the output values are normalized to probabilities using the *softmax* function [12]:

$$p_k = \frac{e^{y_k}}{\sum_{j=0}^{n_y-1} e^{y_j}} \tag{1}$$

We then consider a node *active* if its probability is above a threshold t, and *inactive* if it is below. When exactly one node is active after the computation, we classify the distribution as belonging to the corresponding protocol. Otherwise, the distribution is assumed to belong to a protocol not considered by the network.

To train the network, we initialize all the weights to random values around zero, and compute the value of the outputs to given distributions. The error is the difference between the probable answer and the expected output, which is known in the training phase. The back propagation algorithm using gradient descent and a cross-entropy log loss function is used to adjust the weights. Once sufficient training samples are used to bring the network to optimal performance, we use the network in the feed forward mode to predict the result.

III. Network Processor Implementation

The main challenge in implementing artificial neural network algorithms on network processors is the lack of mathematical operations other than integer addition and subtraction. While multiplication and complex functions could be emulated, the performance would suffer considerably. For this reason, we have resorted to a number of approximations and incremental techniques to achieve results close to the reference implementation, while using only additions.

Before discussing these aspects, we briefly review the architecture of the Mellanox NP-5 network processor [3] that we use in our work, and our operational flow. A simplified block diagram is shown on Figure 2, and consists of five stages that process incoming packets operating in pipeline. These stages are complemented by two data structures. The

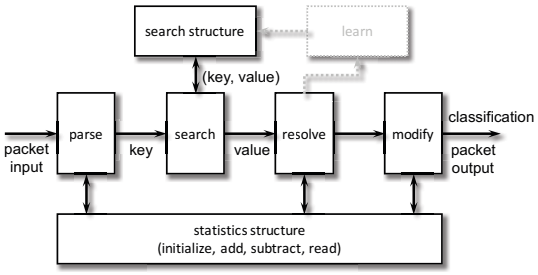

Fig. 2. Simplified architecture of the NP-5 processing pipeline

search structure contains tables made of pairs (key, value) which can be quickly accessed by the `search` stage. The *statistics* structure contains counters which can be initialized, incremented, decremented or read by all stages. Operationally, we store the value of the neurons as counters in the statistics structure and the weights between the input and the hidden layer as a table in the search structure. We use counters for the neurons because they function as accumulators, and they can be stored in internal memory for fast access. We therefore update the state of the whole neural network incrementally for each incoming packet, thus avoiding the use of the multiplication operation, which is not available in the NP-5 architecture.

The flow of operations works as follows. The `parse` stage parses the content of each incoming packet, extracts the packet length information and determines the index i of the bin, and therefore the counter (input neuron), that must be incremented to update the packet size distribution. The same index i is sent to the `search` stage to retrieve the weights between input neuron i and the hidden layer. Notice that input neuron i was incremented by 1, while all other inputs have not changed. Therefore, the hidden layer neurons must be incremented exactly by the amount of the weights incoming from input neuron i, while the other weights can be ignored. This operation, executed in the `resolve` stage, uses only addition and subtraction (for positive and negative weights) and avoids the use of the multiplication.

The `resolve` and `modify` stage then process the rest of the network, by computing the activation function of the hidden layer, and updating the values of the output layer counter. The weights between the hidden and the output layer are stored as constants directly in the code, instead of using

the search structure, since they must all be used for every packet. The stages then apply the softmax criterion to classify the packet distribution. The `learn` stage can be used to dynamically update the search structures, and is not currently used in our implementation. The NP-5 includes 64 pipelines of this kind to increase the processing rate.

The `resolve` and `modify` stages employ a number of approximations which are discussed below.

A. Fixed point value representation

Since the NP-5 does not provide native support for floating point numbers, we have used a fixed point representation. Looking at the training results, we observe that the weights between the input and the hidden layer are confined within the interval $[-1 : 1]$, while those between the hidden and the output layer lie within $[-2 : 2]$. We therefore use a sign and magnitude fixed point representation for the weights, where 1 bit is dedicated to the sign, 2 bits to the integer part, and the rest to the fractional part. Using a total of 16 bits, the fractional part is represented by 13 bits, giving a resolution of $\pm 2^{-13} = 0.00012$, which is sufficient for our aims. We use a sign and magnitude representation for the weights because the NP-5 does not support two's complement operations directly, and the sign tells us immediately whether we need to use addition or subtraction, while the magnitude provides the amount without having to use a conversion. We instead use a two's complement representation for the neuron values, which are stored in the counters, because this allows us to apply the operations without having to look at their sign. These values use the same number of fractional bits as the weights, however the integer part must be much larger, since they accumulate updates from different packets. Fortunately, counters are 96-bit wide, so we have space to spare. To mimic the two's complement encoding, we simply initialize all hidden and output neurons to an offset of 2^{20}: values above 2^{20} are considered positive, values below are considered negative and in magnitude equal to the complement to 2^{20}.

B. Activation function

When training the network, it is essential to use the hyperbolic tangent (or another differentiable function) as activation function for the hidden layer to be able to compute the gradient and update the weights. Network evaluation, on the other hand, does not have this constraint. We therefore approximate the hyperbolic tangent using the sign function. This has two advantages. First, the sign function involves only a comparison with zero, thus avoiding the intricacies of computing a complex function. Secondly, the result of the sign function is either 1 or -1. Recall that, in order to compute the output layer, we must multiply the hidden to output weights w^{hy} by the corresponding hidden layer values. Since these can only be 1 or -1, the multiplication reduces to simply a possible change of sign. The remaining operations are additions. The results show that this approximation works well, especially since random errors from different nodes tend to cancel out when added together. Thus, for every packet, we re-initialize the output layer to zero, and compute their new values by considering, in turn, the contribution of each of the hidden nodes. Unlike the computation from input to hidden layer, this cannot be done incrementally, since several hidden neurons may change at the same time, thus complicating the update. In this case, recomputing the values is simpler.

Notice that we do not overwrite the hidden neurons with their activated value, but we instead use it immediately to update the output layer, and then discard it, since it is of no further use. This way, the hidden neurons retain their actual, non-activated, value in order to correctly perform the incremental update from the input layer upon reception of the next packet.

C. Softmax approximation

The final classification step consists in normalizing the output layer values, followed by a threshold operation to choose the active output. As discussed, normalization is traditionally implemented using the *softmax* function according to equation (1), which results in a set of probabilities. After normalization, the value of each neuron of the output layer is saturated to 1 if it is above a threshold t, and to 0 if it is below. In our case, we have chosen $t = 0.1$. A valid classification is obtained if only one of the output neurons is above the threshold. In all other cases, the distribution is considered unknown.

To avoid the use of the exponential function, we have determined the conditions that the original neuron values must satisfy, before normalization, so that only one neuron is above the threshold after normalization. Without loss of generality, assume y_0 is the largest of the output neurons, and that y_1 is the second largest. Because normalization is monotonic, in order to have only one active output we must impose that $p_0 \geq 0.1$ and that $p_1 \leq 0.1$. From the condition $p_0 \geq 0.1$ and equation (1), rearranging the terms and solving for y_0, we obtain:

$$y_0 \geq \ln(0.1/0.9) + \ln\left(\sum_{j=1}^{n_y-1} e^{y_j}\right) \tag{2}$$

We take advantage of the following relations, which are easy to prove, to compute bounds for the right-hand side:

$$\ln\left(\sum_{j=0}^{n-1} e^{a_j}\right) \geq \max(a_0, \ldots, a_{n-1}) \tag{3}$$

$$\ln\left(\sum_{j=0}^{n-1} e^{a_j}\right) \leq \max(a_0, \ldots, a_{n-1}) + \ln(n) \tag{4}$$

Thus, using equation (4), a conservative condition is that

$$y_0 \geq -2.197 + y_1 + \ln(n_y - 1)$$

This condition is always satisfied in our case, since $n_y = 3$ and by definition $y_0 \geq y_1$.

Starting from $p_1 \leq 0.1$, applying again equation (1) and solving for y_1 we obtain:

$$y_1 \leq \ln(0.1/0.9) + \ln\left(\sum_{j=0, j\neq 1}^{n_y-1} e^{y_j}\right) \tag{5}$$

Since y_0 is the largest, using equation (3) a conservative condition is

$$y_1 \leq -2.197 + y_0 \tag{6}$$

Thus, if $y_0 \geq y_1 + 2.197$ then certainly $p_0 \geq 0.1$ and $p_1 \leq 0.1$, and we have a valid classification. Because of the conservative approximation, the converse is not always true, and less stringent conditions could be derived. For instance, using similar arguments, we can show that if y_0 is the largest value and all other output neurons are the same value, then it is sufficient that $y_0 \geq y_1 + 2.08$ in case of 3 output neurons. This is actually the best case. Our approximate condition (6) is therefore not too tighter, and we expect it to behave correctly most of the times. This condition involves only additions, and can be easily implemented on the network processor. Because, naturally, the neurons are not ordered, we need to apply the check to all pairs of output neurons.

IV. Experimental results

We have evaluated our implementation comparing accuracy and performance of the NP implementation against a C language implementation of the floating point version, with full mathematical support, and a fixed-point C version using the same approximation techniques as the NP (sign activation function and simplified softmax). For network training and testing, we have generated traffic under all the investigated protocols and collected the packet size distributions using Wireshark. In particular, for the generation of MQTT traffic we have used the *MQTT Python client library*[1], for CoAP we have used the Phyton library *CoAPthon*[2], and for AMQP we have used the Phyton library *Pika*[3].

We have used 13 distributions for training, and 16 distributions comprising 2193 packets for testing, which are correctly classified by the neural network. In our evaluation we are more interested in the performance of the approximations. Table I reports the average error and standard deviation between the approximated and the floating point versions at the *hidden layer*, after applying the *activation function* and at the *output nodes*. The results show that the error is small. Indeed, the approximated neural network classifies all the test distributions exactly as the floating point version. This shows that the approximations are adequate, while the sign function does not alter the activation function appreciably.

TABLE I
Error estimation

	Hidden		Activation		Output	
	err.%	std.dev.	err.%	std.dev.	err.%	std.dev.
MQTT	0,02	0,0001	1,17	0,0240	1,22	0,0218
AMQP	0,03	0,0002	0,01	0,0003	0,01	0,0002
CoAP	0,04	0,0003	0,49	0,0081	2,57	0,0267

On the other hand, performance is substantially better for the approximated version. Table II shows the throughput achieved by the different versions, using the incremental update in all cases, in terms of packet rate (Millions of Packet per second). The C versions, both floating point and approximated, are executed over Ubuntu on an Intel i7 quadcore PC at 2.5 GHz. The NP version is executed on the NP-5, running at 500 MHz with 64 parallel pipelines. The results show that the approximated C version runs 4.3 times faster than the floating point version, while the NP implementation runs 14 times faster, and is over 3 times faster than the approximated C version. On top of that, the NP-5 version includes packet acquisition, inspection, parsing and forwarding, which are not accounted for in the C implementation, and which could considerably impact performance. Assuming packets are longer than 127 bytes, the NP-5 can handle a 100 Gbps channel.

TABLE II
Performance comparison

	C (floating)	C (approx.)	NP-5
Throughput	7.04 MPps	30.3 MPps	98.5 MPps

V. Conclusion

We have developed a neural network packet classifier for the IoT running at 100 Gbps on a network processor. We have discussed a set of approximations to make the implementation feasible and to improve performance. Given the increasing adoption of machine learning at the network level, these techniques may find applications in other fields, such as recognition and diagnosis and data analysis. These aspects are part of our future work.

[1] https://pypi.python.org/pypi/paho-mqtt/1.2
[2] https://github.com/Tanganelli/CoAPthon
[3] https://pypi.python.org/pypi/pika

References

[1] T. T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.
[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communication Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.
[3] O. E. Ferkouss, I. Snaiki, O. Mounaouar, H. Dahmouni, R. B. Ali, Y. Lemieux, and C. Omar, "A 100Gig network processor platform for OpenFlow," in *Proceedings of the 7th International Conference on Network and Service Management*, Oct. 2011.
[4] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400G+ packet classification," in *Proceedings of the 13th International Conference on High Performance Switching and Routing*, June 2012.
[5] M. E. Kounavis, A. Kumar, H. Vin, and R. Yavatkar, "Directions in packet classification for network processors," 2003.
[6] A. Dainotti, A. Pescape, and K. C. Claffy, "Issues and future directions in traffic classification," *IEEE Network*, vol. 26, no. 1, January 2012.
[7] P. Ohm, D. Sicker, and D. Grunwald, "Legal issues surrounding monitoring during network research," in *In Proceedings of the 7th ACM SIGCOMM Conf. on Internet Measurement*, 2007, pp. 141–148.
[8] A. Bär, P. Svoboda, and P. Casas, "MTRAC - discovering M2M devices in cellular networks from coarse-grained measurements," in *Proceedings of the International Conference on Communications*, June 2015.
[9] L. Grimaudo, M. Mellia, E. Baralis, and R. Keralapura, "SeLeCT: Self-learning classifier for internet traffic," *IEEE Transactions on Network and Service Management*, vol. 11, no. 2, pp. 144–157, June 2014.
[10] F. Shen, C. Pan, and X. Ren, "Research of P2P traffic identification based on BP neural network," in *Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Nov 2007.
[11] R. Gopalapillai, J. Vidhya, D. Gupta, and T. S. B. Sudarshan, "Classification of robotic data using artificial neural network," in *IEEE Recent Advances in Intelligent Computational Systems*, Dec 2013.
[12] H. Trussell, A. Nilsson, P. Patel, and Y. Wang, "Characterization, estimation and detection of network application traffic," in *Proceedings of the 13th European Signal Processing Conference*, Sept. 2005.