EFFICIENT TOPOLOGY CODING AND PAYLOAD PARTITIONING TECHNIQUES FOR NEURAL NETWORK COMPRESSION (NNC) STANDARD

Jaakko Laitinen[*], *Alexandre Mercat*[*], *Jarno Vanne*[*],
Hamed Rezazadegan Tavakoli[†], *Francesco Cricri*[†], *Emre Aksu*[†], *Miska Hannuksela*[†]

[*]Ultra Video Group, Tampere University, Tampere, Finland
[†]Nokia Technologies, Tampere, Finland

ABSTRACT

A Neural Network *Compression* (NNC) standard aims to define a set of coding tools for efficient compression and transmission of neural networks. This paper addresses the high-level syntax (HLS) of NNC and proposes three HLS techniques for network topology coding and payload partitioning. Our first technique provides an efficient way to code prune topology information. It removes redundancy in the bitmask and thereby improves coding efficiency by 4–99% over existing approaches. The second technique processes bitmasks in larger chunks instead of one bit at a time. It is shown to reduce computational complexity of NNC encoding by 63% and NNC decoding by 82%. Our third technique makes use of partial data counters to partition an NNC bitstream into uniformly sized units for more efficient data transmission. Even though the smaller partition sizes introduce some overhead, our network simulations show better throughput due to lower packet retransmission rates. To our knowledge, this the first work to address the practical implementation aspects of HLS. The proposed techniques can be seen as key enabling factors for efficient adaptation and economical deployment of the NNC standard in a plurality of next-generation industrial and academic applications.

Index Terms— Neural Network Compression (NNC), Neural Network Representation (NNR), High-Level Syntax (HLS), neural network topology, bitmask coding

1. INTRODUCTION

The proliferation of intelligent applications and their multifaceted set of machine learning algorithms have fueled the need for deep *neural networks* (*NNs*) that tend to be very resource hungry. At the same time, the increasing sizes of advanced deep NNs give rise to deployment costs that stem from their higher computation load, memory consumption, and bandwidth utilization. To that end, there is a strong industrial demand for more compact deep NNs.

The two main objectives of NN coding are 1) to create a compact, high-performance NN model that can be run on resource-limited devices; and 2) to enable efficient NN transport that is the key to massive deployment of edge computing and frequent updates of NN models in remote cloud services. These two objectives can be addressed independently, but depending on the use case, they can also complement each other.

Research on NN compression is still underway, but the urgent industrial demand has already resulted into the first draft standard called *Neural Network Compression* (*NNC*) or ISO/IEC FDIS 15938-17 [1]. It defines a set of coding tools and techniques for compressing and decompressing NNs for efficient transportation. The NNC reference software is called *Neural Network Compression Test Model* (*NCTM*) [2]. It aims to implements all normative coding tools of NNC, so it is typically used as an experimental software.

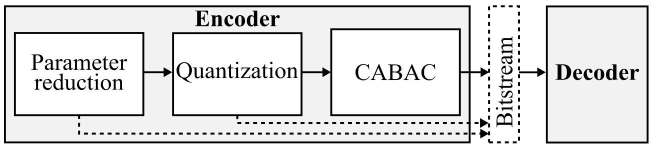


Fig. 1. High-level description of the NNC encoding tools.

Figure 1 depicts a high-level block diagram of the proposed encoding scheme, where the proposed tools are included in the parameter reduction step. The remaining steps include quantization, used to prepare model weights for the encoding process as well as improve coding efficiency, and entropy coding in the form of CABAC [3]. The standard specifies the compressed representations for NNs and creates a compatible bitstream for the NNC decoder. In order to signal information for the decoding operations, the NNC standard defines a *high-level syntax* (*HLS*) and a set of elements that carry the information of employed tools at the encoding step.

The NNC bitstream is composed of *NN Representation* (*NNR*) units that are divided into unit size, unit header, and unit payload fields. The first field gives the total size of the unit. The unit header indicates the type of the carried information and associated metadata. For example, an *NNR start* unit indicates the start of a bitstream, a *Model Parameter Set* (*MPS*) unit carries the global metadata of an NN, a *Layer Parameter Set* (*LPS*) unit is for the layer level information and can contain partial representations of NNs, a *Topology* (*TPL*) data unit carries topological information of an NN, *compressed data* (*NDU*) units contain NN model weights, etc. The third field, unit payload, includes actual NN data. Throughout the HLS, element IDs are used to refer to individual NN layers. The NNC standard additionally defines two different modes for the IDs: a string format label and a 0-indexed integer ID that also defines an ordering for the layers in the NN model. If the integer-based IDs are used, the layer topology needs to be provided separately. A more detailed HLS description of the NNC standard can be found in [1], [3], but they do not address implementation aspects.

To the best of our knowledge, this is the first work to propose practical HLS implementation techniques for the NNC standard. Our primary focus is on a compact representation of prune topology information, efficient bitmask coding, and bitstream partitioning with partial data counters.

The remainder of this paper is organized as follows. Section 2 introduces the proposed techniques in detail. Section 3 reports our experimental results. Finally, Section 4 concludes the paper.

This work was supported in part by the AI for situational Awareness (AISA) project led by Nokia and funded by Business Finland.

2. PROPOSED CODING TECHNIQUES FOR NNC

The proposed HLS techniques include 1) a compact prune bitmask for higher coding efficiency; 2) sparse bitmask coding in chunks for lower coding complexity; and 3) partial data counters for more economical transmission. These techniques are detailed next.

2.1. Prune Bitmask

Prune bitmasks are metadata included in the optional HLS element called prune topology container. They allow passing information about the pruning process and provide a decoder with a way to determine the post-pruning structure of an NN model. Alternatively, the pruning topology container allows duplicating the pruned structure of the original model when structure changes take place.

Prune bitmasks are generated in the parameter reduction step (see Figure 1) as a way to reduce the complexity of an existing NN. In the pruning process, input and output channels of convolutional layers are removed according to the specified criteria, such as estimated weight importance and a pruning ratio [3]. However, the input channels of the first layer and the output channels of the last layer are left unchanged. Additionally, linear layers connected to the pruned convolutional layers need to be pruned to reflect the removed connections. A straightforward way to generate a prune bitmask is to assign each weight of a layer a value of one if the weight should be pruned and zero otherwise. However, this solution contains a lot of redundancy as only the pruned input/output channels are relevant, not the specific weights that are pruned.

The proposed technique for redundancy removal is referred to as *partial prune bitmask* (*PPB*). It removes redundancy by only generating a bitmask for the output channels of the layers. The input channels that should be pruned can be inferred from the bitmask of the previous layer, since the output and input channel sizes of connected convolutional layers should match. If the model contains linear layers after the convolutional layers, the input channels of the first linear layer need to be pruned while taking into account that the linear layer may have more inputs than the convolutional layer has outputs. In this case, the ratio of outputs to inputs is calculated and the linear input channels are pruned accordingly. Finally, if the last output layer is a convolutional layer, it is omitted from the prune bitmask, since the final layer is not pruned, as mentioned above.

2.2. Packed Coding of Sparse Bitmasks

Sparse bitmasks are also included in the optional prune topology container. They allow transmitting information that can be used as side-channel information, e.g., to zero the weights of the original model to match to the sparsified model without transmitting the entire sparsified model. Unlike prune bitmasks, sparse bitmasks are specified for all weights of an NN model. This results in massive sparse bitmasks for large models.

As with pruning, sparsification is a part of the parameter reduction step. Sparsification is performed using an iterative process on a pre-trained model with a target sparsification ratio and includes a special loss function for fine-tuning the sparsified model to improve coding efficiency [3].

When generating a bitmask, a value of zero or one is associated with each weight of a parameter tensor. This bitmask of zero/one values is then encoded as 1-bit unsigned integers. By default, the bitmask is encoded one bit at a time. As such, coding each bitmask bit individually is inefficient and result in many function calls when coding bitmasks for large models with millions of weights.

To solve this, we propose coding the bitmask in chunks of N-bits. Because the bitmask is coded as a continuous sequence of bits in the bitstream, the number of simultaneously coded bits has no effect on the bitstream. However, coding multiple bits in one operation increases the coding speed because it reduces the number of required operations. The speedup scales linearly as a function of N and saturates as N exceeds the maximum integer precision of the system. In that case, a bitmask chunk cannot be processed with one operation. The most typical values for N are 8, 16, 32, and 64 since those are the most widely supported integer precisions. Our solution also reduces the run-time memory consumption by packing the bitmask bits directly to N-bit integers, provided that the memory elements can take up N bits.

Ideally, the number of bits per operation can be freely set. In this case, the bitmask coding is carried out by looping over the bitmask in N-bit chunks and the remaining bits (if any) are coded in the final iteration. If the number of bits per operation cannot be dynamically set, the leftover bits must be processed separately, e.g., one bit at a time, i.e., it adds some processing overhead.

Finally, it is possible to perform both pruning and sparsification in which case both a prune bitmask and a sparse bitmask is generated. Moreover, the packed coding technique, described above, can be applied to the prune bitmask as well. To avoid sparsifying parameters that are pruned, the pruning step is performed first and sparsification is performed on the pruned model.

2.3. Partial Data Counter

Partial data counter is an HLS element that is used to track partial NDU NNR unit payloads. It allows splitting large payloads into smaller, more manageable ones. Because partitioning using the partial data counter is done on the bitstream level, there is no need to account for it when designing and training the NN model.

Partitioning large NDUs is especially useful when transmitting an encoded NN model over a network, since large packets are more vulnerable to packet losses and transmission errors, resulting in wasted bandwidth due to retransmitted packets. The downside of fine-grain partitioning is the overhead introduced by the additional NNR unit headers, so balancing is needed between NNR unit and bitstream sizes.

The partitioning is created by first coding an NNR unit payload normally. Then, a sub-payload size is determined as per the desired number of partitions or target NNR unit size. Each sub-payload is written to the bitstream with an NNR unit header and the respective partial data counter value. If a partial data counter value is present in the header, a decoder reads NNR units until a partial data counter value of zero is reached. The payload of each NNR unit is extracted, and the payloads are combined to allow for typical decoding.

Let us define a 'max_ndu_nnr_unit_size' value that is used to control the partial data counter value on a per-NDU basis. It allows for a uniform maximum size across the NDU NNR units. First, the encoder determines an initial estimate for the NDU payload partition size. A mock encoding is performed to determine the header size that is then used to calculate the correct ratio between the header and the payload. Because the size of an NNR unit header can change depending on the total size of the NNR unit, care needs to be taken when the 'max_ndu_nnr_unit_size' value is close to the break-even point, where the NNR unit header size switches from short to long.

Table 1. Breakdown of the executed NN models

NN model	Total param	Conv. param	Total layer	Conv. layer
UC12B	76179	76179	9	9
DCase	116118	102016	3	2
MobileNetV2	3538984	2189760	53	52
ResNet50	25636712	23398976	50	49
VGG16	138357544	14714688	16	13

3. EXPERIMENTAL RESULTS

Our experiments were carried out with NCTM [2] version 5.0 that was modified to support the proposed optimization techniques. For tests that perform encoding/decoding, the following NCTM parameters were used: approximation (i.e., quantization) uses the 'codebook'-method [3] with a 'cb_size_ratio' of 5000 and 'q_mse' of 0.00001. All tests were performed on a computer equipped with Intel Xeon @ 3.70 GHz 8-core processor and 32 GB of RAM.

Our test set is characterized in Table 1. Altogether, it includes five NN models taken from the evaluation specification for NNC [4]. UC12B and DCase are smaller models for autoencoding and acoustic classification, respectively. MobileNetV2 [5], ResNet50 [6], and VGG16 [7] are large image classification models.

3.1. Partial Prune Bitmask

The proposed PPB technique is compared with 1) a full prune bitmask that covers all weights of a model; 2) prune dictionary using string-based labels for element IDs [1]; and 3) prune dictionary using index-based labels. With each model, the sizes of prune bitmasks are estimated from the number of weights and layer output channels of the convolutional layers. The difference between the solutions is given as *compression gain* (*CG*) that is defined as

$$\mathrm{CG} = \left(1 - \frac{\mathrm{S}_1}{\mathrm{S}_2}\right) \times 100, \qquad (1)$$

where S_1 and S_2 are the sizes of the two HLS elements.

Table 2 shows the results. Using the partial bitmask results in a CG of over 99% for all tested models. Against index-based prune dictionaries, the proposed prune bitmask overall provides better compression but, depending on the NN model, CG ranges from 4% to 95%. Compared to the string label prune dictionary, prune bitmask gives a CG ranging from 60% to 98%. However, the sting label sizes depend on the lengths of the strings used as the labels, so the exact CG may vary. With the larger models, especially VGG16, the size of indexed labels gets close to the size of the PPB. This is due to the relatively low number of layers compared to the number of weights, especially in the fully connected layers (see Table 1).

3.2. Packed Coding of Sparse Bitmasks

The coding complexity of a sparse bitmask is calculated by measuring the time it takes to encode/decode topology information with and without the proposed optimizations. Majority of the complexity comes from the bitmask coding, so the reported times closely reflect the true coding time for the bitmasks. Time measurements are repeated 20 times for more reliable results.

The encoding and decoding times for sparse bitmask coding (packed and unpacked) are shown in Table 3. The speedup ranges from 2.54× to 2.87× in encoding and from 5.24× to 5.56× in decoding. However, the theoretical speedup of 8× is still out of reach because the underlying bitstream generation is executed one bit at a time. The proposed packing only moderately reduces the number of total function calls. For decoding, this reduction in the number of function calls provides a more significant speedup.

Table 2. Compressibility of prune bitmask

NN model	Prune bitmask (full)	CG of PPB	CG of PPB vs. string label	CG of PPB vs. indexed label
UC12B	75888	99.62%	94.68%	88.77%
DCase	101920	99.91%	98.54%	94.64%
MobileNetV2	2189760	99.22%	87.65%	43.48%
ResNet50	14710464	99.97%	77.36%	25.89%
VGG16	23454912	99.89%	60.60%	4.35%

Table 3. Coding time of sparse bitmasks

NN model	Encoding time (s)			Decoding time (s)		
	Unpacked	Packed	Speedup	Unpacked	Packed	Speedup
UC12B	22.0	7.8	2.83×	8.6	1.6	5.49×
DCase	34.0	13.4	2.54×	12.7	2.4	5.24×
MobileNetV2	1011.8	353.1	2.87×	395.5	71.1	5.56×
ResNet50	7462.0	2762.3	2.70×	2871.7	534.1	5.38×
VGG16	41229.0	16092.0	2.56×	15237.3	2866.3	5.32×

For further speedup, the underlying coding process could be accelerated with dedicated functions that are able to code larger HLS elements with a single function call. For example, coding the sparse bitmask in 8-bit chunks would reduce function calls to one eighth.

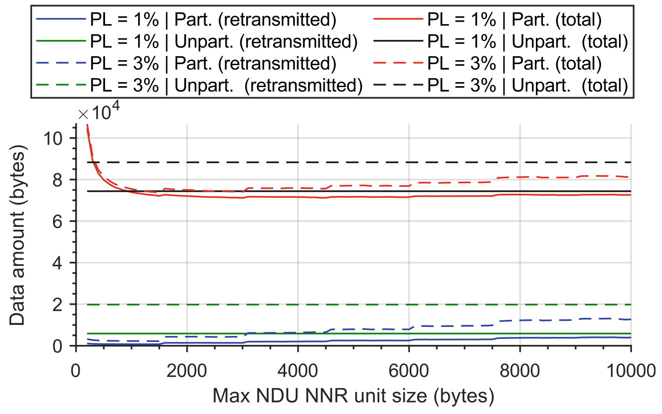
3.3. Partial Data Counter

Partial data counter results were obtained with network simulation. First, a bitstream was generated using a partitioning determined by a given 'max_ndu_nnr_unit_size' value or without any NNR unit partitioning. Then, the bitstream was decoded and the number and size of the NDU NNR units were extracted for the network simulation. A packet loss probability was also specified to simulate lost packets and transmission errors in the network simulation.

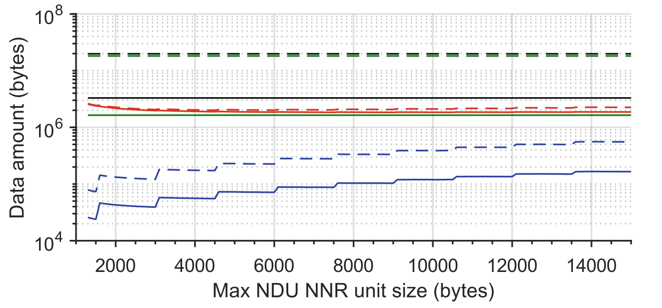
A single simulation run consists of sending each NNR NDU unit (e.g., a packet) in the generated bitstream. Packet sending is simulated using a randomized function, returning true or false with the specified probability, to determine if a packet is delivered or not. If the sending failed, the packet was retransmitted, and relevant statistics were updated. The output of the simulation run is the amount of (re)transmitted data. These simulation runs were repeated 100 000 times, and the results were averaged to get the expected values (of the random event) for the given parameters. A *maximum transmission unit* (*MTU*) size of 1500 bytes was also specified to better reflect real world network conditions. If a packet being sent was larger than the MTU size, it was further divided into subpackets that were sent individually. If any of the subpackets were lost, all subpackets were retransmitted. The network simulation assumed a transport protocol with minimal packet management such as UDP.

Figure 2 plots the data transmission rates for different max NNR NDU unit size values in the network simulation with *packet loss* (*PL*) values of 1% and 3%. UC12B and MobileNetV2 NN models were chosen to cover both the small and large NN cases. For MobileNetV2, *fully connected (FC)* layers were omitted in our tests because they form the largest NDUs, and transmitting the unpartitioned bitstreams without proper packet management becomes impossible even with a small number of packet losses.

In Figure 2, data rates and NNR unit sizes are given in bytes. The "Unpart." stands for the unpartitioned NNR NDU units and the "Part." for partitioned NNR NDU units. The lines with "(total)" show the total amount of data that has been sent, whereas the



(a) UC12B



(b) MobileNetV2 (no FC layer)

Fig. 2. Network simulation results for UC12B and MobileNetV2.

"(retransmitted)" suffixed lines only show the amount of data that has been retransmitted.

In Figure 2(a), we can see that the partitioned bitstream becomes more efficient than the unpartitioned bitstream when the max NNR unit size is 1000 bytes or more (packet loss of 1%) with the UC12B model. However, the total data amount is still fairly similar to the unpartitioned case. With a packet loss of 3%, the better efficiency of the partitioned bitstream becomes clearer. Even smaller max NDU NNR unit size limits are more efficient than the unpartitioned bitstream due to the increase in the retransmission amount of the unpartitioned bitstream with higher packet loss values. With UC12B, we can see that going over the MTU size still decreases the total transmission amount for a bit before it starts increasing again as the retransmission amount increases. This dip is due to the considerable increase in bitstream size, caused by a large number of NDU partitions, that overpowers the increase in the retransmission amount, brought on by larger packets. For UC12B, the smallest total transmission amount is around 3000 bytes, for a packet loss of 1%. However, if the packet loss is 3%, the optimal NDU size is the MTU limit of 1500 bytes.

For the MobileNetV2 results in Figure 2(b), the large NDU unit sizes call for larger maximum NDU NNR unit sizes, starting from the MTU size of 1500 bytes. From the data, the optimal maximum NDU NNR unit size for MobileNetV2 is around 9000 bytes, when packet loss is 1%, but increasing the packet loss to 3% reduces the minimum to around 4500 bytes. As for the unpartitioned bitstream, its total transmission amount is higher than the respective partitioned bitstream, for both packet loss values, throughout the NDU NNR unit size range.

4. CONCLUSION

In this paper, we proposed techniques for creating a compact prune bitmask representation, coding sparse bitmasks efficiently, and partitioning NNR NDU units economically.

First, limiting the prune bitmask to output channels provides a very compact way to store prune topology information, with coding efficiency ranging from 4% to 99% over other methods. However, this approach may become difficult to implement with large NN models and complex features (e.g., bottlenecks, residual blocks, and shortcuts) that do not necessary translate well to NNR. In those cases (e.g., image classification), the prune dictionary representation is a more straightforward approach and, depending on the model, can be implemented with a slight coding overhead when compared with the prune bitmask. In addition, the more compact prune information representations decrease the decoding complexity, and the pruning process can be implemented as a slice operation on the parameter tensors without increasing the complexity of the overall pipeline.

Secondly, sparse bitmasks become very complex with large NN models, and their coding time can rise to tens of hours in the worst case. Using the proposed method of coding the bitmask in chunks reduces the NNC encoding time to almost one-third and the decoding time to over one-fifth.

Finally, our experiments showed in simulated network conditions that transmitting full NNR NDU units over a network becomes prohibitively inefficient as the probability of packet losses increases. Using the partial data counters in NDU partitioning comes with some coding overhead over that of the unpartitioned bitstream, but it is well compensated by lower packet retransmission rate as the partition size grows. For the examined NN models, the optimal NDU size is 2–6× the MTU size with a packet loss of 1% and 1–3× the MTU size with a packet loss of 3%.

5. REFERENCES

[1] *Text of ISO/IEC FDIS 15938-17 Compression of Neural Networks for Multimedia Content Description and Analysis*, ISO, 2021.

[2] *Test Model 6 of Compression of Neural Networks for Multimedia Content Description and Analysis*, MPEG document N0017, ISO/IEC JTC 1/SC 29/WG 04, Oct. 2020.

[3] H Kirchhoffer et al., "Overview of the neural network compression and representation (nnr) standard," *IEEE Trans. Circuits Syst. Video Technol.* , Jul. 2021, pp. 1–14.

[4] *Evaluation Framework for Compressed Representation of Neural Networks*, MPEG document N17929, ISO/IEC JTC 1/SC 29/WG 11, Oct. 2018.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vision Pattern Recogniti.*, Jun. 2018, pp. 4510–4520.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vision Pattern Recogniti.*, Jun. 2016, pp. 770–778.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, Sept. 2014.