https://doi.org/10.2352/EI.2024.36.17.AVM-113

Efficient fault tolerant architecture for neural network compute

Shyam Jagannathan, Mihir Mody, Prithvi Shankar, Villarreal Jesse, JuneChul Roh, Kumar Desappan, Deepak Poddar, Pramod Swami, Embedded Processors Business, Texas Instruments

Abstract

With artificial-intelligence (AI) becoming the mainstream approach to solve a myriad of problems across industrial, automotive, medical, military, wearables and cloud, the need for high-performance, low-power embedded devices are stronger than ever. Innovations around designing an efficient hardware accelerator to perform AI tasks also involves making them fault-tolerant to work reliability under varying stressful environmental conditions. These embedded devices could be deployed under varying thermal and electromagnetic interference conditions which require both the processing blocks and on-device memories to recover from faults and provide a reliable quality of service. Particularly in the automotive context [1], ASIL-B compliant AI systems typically implement error-correction-code (ECC) which takes care of single-error-correction, double-error detection (SECDED) faults. ASIL-D based AI systems implement dual lock step compute blocks and builds processing redundancy to reinforce prediction certainty, on top of protecting its memories. Fault-tolerant systems take it one level higher by tripling the processing blocks, where fault detected by one processing element is corrected and reinforced by the other two elements. This becomes a significant silicon area adder and makes the solution an expensive proposition. In this paper we propose novel techniques that can be applied to a typical deep-learning based embedded solution with many processing stages such as memory load, matrix-multiply, accumulate, activation functions and others to build a robust fault tolerant system without linearly tripling compute area and hence the cost of the solution.

Introduction

Neural-network (NN) based AI solutions are widely applied to solve multiple real-world problems such as classifying different objects over a conveyor belt, identifying vehicles, pedestrians traffic signs, traffic lights, segmenting free space for autonomous robots to navigate and more, offering a high degree of accuracy and performance. By design, a neural network consists of series of computationally connected layers as shown in Figure 1. Each layer has features and the connections between features are weights which are learnt by training the network. Deep convolutional neural networks [2] (CNN) are particularly applied to image-based input signals coming from a camera to infer real world objects. These computations can be reduced to basic matrix multiply accumulate operations which makes hardware design simple as opposed to traditional computer vision techniques which required specialized processors to implement custom mathematical operations on a hardware accelerator or a Digital Signal Processor (DSP). [3]

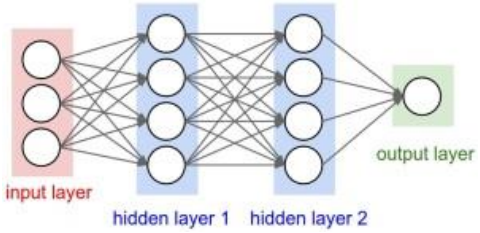

Figure 1. A typical neural network with one input layer, two hidden layers and one output layer. Circles represent features and arrows represents trained weights.

While designing neural network based embedded solutions multiple optimization vectors need to be considered as shown in Figure 2. Performance of a NN processing block is measured in terms of Trillion-Operations-Per-Second (TOPS). Embedded device constrains of power [4] and cost also gets measured as TOPS/watt and TOPS/dollar respectively. Software scalability and ease-of-use also influences the hardware design decisions to make it easy for customers to program the device. Other important vectors are security, safety, system performance and multi-modality [5]. In this paper we will discuss the challenges of building an embedded device which can work safely and reliably under stressful environmental conditions of varying temperature changes, electromagnetic interference which can affect the memories and processing block of the compute unit.

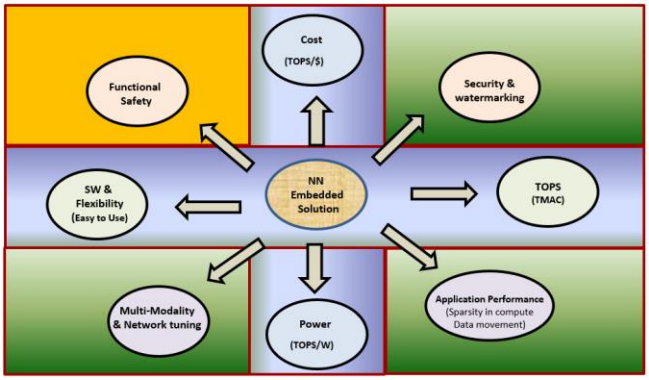

Figure 2. Various optimization vectors to be considered while designing an embedded processor for running neural networks

A typical embedded device implementing a processing block is systolic [6] by design. There is a processing block which reads the operands as an input and writes the result as an output to a memory block. CNN processing can be implemented as simple matrix multiply operation of trained weights and features followed by accumulation of partial results at every stage. The final accumulated result is passed through an activation function which helps normalize the output within a certain range. Large matrix multiplications are broken down as a sum of product of smaller matrices. Inner- product based matrix multiply accumulate units could be designed as shown in Figure 3 where the row vector A [1 x M] is multiplied by matrix B [MxM] producing C matrix [MxM] of results. The A row vector could directly fetch weights from memory whereas the B matrix could be double buffered to parallelize load and compute operation. The C matrix is also double buffered to parallelize accumulate and store operations. The store path of C result can have an activation function which adjust the output range before its stored into the memory.

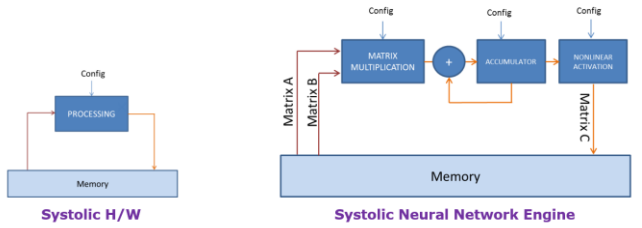

Figure 3. Systolic neural network engine comprising of key stages, matrix A and B read from memory, matrix multiplication, accumulation and non-linear activation and write of matrix C back to memory

Faults can occur at any stage in the operation affecting memory and processing elements. In the automotive context, for ASIL B safety levels the memory is protected with parity or ECC [8]. The processing element is checked for faults with a known test pattern in a programed time internal called Built-In-Self-Test [7] (BIST) mechanism. For ASID-D levels dual-lockstep processing elements are used where fault in one is detected by other. For fault tolerant systems a minimum of triple-lockstep processing elements is deployed where fault in one is detected by the other two and corrected. While this makes the solution robust and fault tolerant, it also becomes expensive as it linearly adds area and hence the cost of the device.

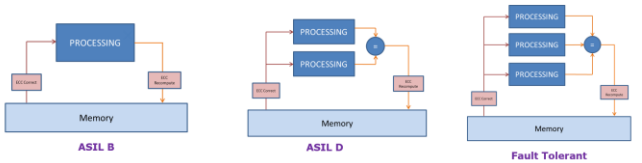

Figure 4. In automotive safety, ASIL B systems have ECC/parity memory protections, ASIL-D systems have memory protections and compute protection with dual lockstep processing. Fault tolerant system have triple redundancy where fault in one processing block is detected and corrected by other two processing blocks along with memory protections

A reliable fault tolerant neural network system can be built by breaking down the processing block into smaller processing stages and reinforcing the data fetch and data compute integrity at each stage as shown in Figure 5. The below proposal assumes the size of participating matrices of size MxM with fixed-point integer elements of 8-bit depth. However, it can be extended to apply for any arbitrary matrix size and element bit-depth. The illustration also shows row and column location of a potential fault which can be corrected by algebraic corrections. These computations of detection and correction has to occur in parallel and ahead of actual compute to maintain real-time performance.

Stage1: Correct fault in the input data

Compute and store row and column checksums of A and B matrix in the memory a-priori. A column checksum could simply be the sum of all elements along a matrix column. A row checksum is the sum of all elements along a matrix row. While loading the A and B matrix during inference, compute the row and column checksums and compare it against the pre-computed checksums fetches via separate path. Any mismatch will indicate the exact faulty location which can be corrected. Let A be the input matrix of size [MxM] with elements ranging from (a_{11} to a_{mm}), B be the input matrix of size [MxM] with elements ranging from (b_{11} to b_{mm}). Let A_c and B_c be the column sums and A_r, B_r be the row sums of A and B matrix,

$$A_r[i]_{Mx1} = SUM\ (a_{ij}),\ B_r[i]_{Mx1} = SUM\ (b_{ij})$$

$$A_c[j]_{1xM} = SUM\ (a_{ij}),\ B_c[j]_{1xM} = SUM\ (b_{ij})$$

Stage2: Correct fault in the matrix multiply stage

A matrix [MxN] times row checksum of B matrix [Nx1] produces row checksum of C matrix [Mx1]. Similarly, column checksum of A matrix [1xN] with B matrix [NxM] produces column checksum of C matrix [1xM]. Fetch the column and row checksums of C matrix via a separate path and compare against the column and row sum of the computed C matrix [MxM] after multiplying A and B matrix. This will provide the exact location of fault and can be corrected by algebraic operations.

$$C_r[i][Mx1] = A_{MxM}\ x\ B_r[i]_{Mx1}$$

$$C_c[j][1xM] = A_c[j]_{1xM}\ x\ B_{MxM}$$

Stage 3. Correct fault in matrix accumulation stage

The row and column sums of A and B matrix are simply added iteratively to produce the final row and column sums of C matrix. This is compared against the row and column sum of C matrix after final accumulation. This too will provide exact location of fault which can be corrected by algebraic operations.

$$C_r[i]_{Mx1} = A_r[i]_{Mx1} + B_r[i]_{Mx1}$$

$$C_c[j]_{1xM} = A_c[j]_{1xM} + B_c[j]_{1xM}$$

Stage 4. Correct fault in activation functions

Typically, activation functions can be simple rectified-linear-units [10] (RELU) or be non-linear in nature. These often take much less gates when implemented as look-ups or as cordic function units [11]. So, applying triple redundancy logic here does not impact the overall area too much.

Stage 5. Result checksum

After passing the final C matrix result through activation, the row and column checksums are computed and stored in the memory for later stages.

$$C_r[i]_{Mx1} = SUM\ (c_{ij})$$

$$C_c[j]_{1xM} = SUM\ (c_{ij})$$

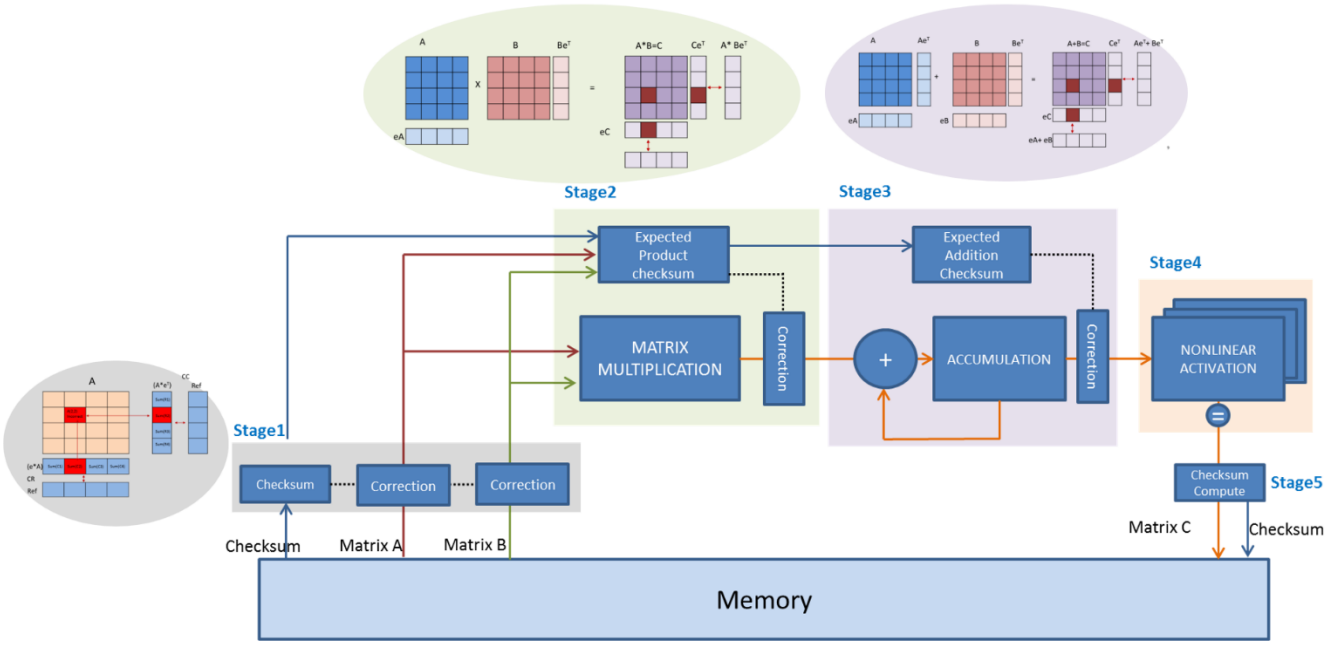

Figure 5. Proposal at various stages of processing with dedicated paths to read reference matrix checksum, calculating checksums on the fly, comparing and correcting errors at every stage. Only the non-linear activation stage is replicated as it does not add to overall area cost significantly.

The proposed solution is simulated assuming a matrix panel size of 64x64 8-bit elements. We need 512-bit bus to fetch 64 8-bit elements in a single clock for each A and B panels. In parallel we fetch 16bit checksums via a 128-bit bus at a slower rate. This is because column and row checksums can be fetched at 1/4th the rate we fetch actual A and B elements. Notice the bit expansion from 8-bit to 16-bit that occurs when we produce column and row checksums of each A and B matrix. The expected checksum is compared against the computed checksum. The row and column position of single or multiple single point errors can be detected and corrected. During the multiply stage the checksums expand to 24bit (8bit x 16bit). A matrix times B row checksum results in C row checksums which are 24-bit in depth. Similarly, B matrix times A column checksum results in C column checksums which are 24-bit in depth. Hence the bus to fetch C reference checksums need to 24-bit wide. The expected checksum is compared against computed checksums and exact row and column location of the fault is identified. The value is corrected via algebraic operations before actual matrix multiplication of A and B matrix. The C matrix error can also be detected by comparing against reference C column and row checksums and corrected before it is sent to accumulation stage, but this requires storing and reading partial row and column sums of C matrix which can increase the read bandwidth. In the accumulation stage depends on how deep the convolution filters are and the accumulated result can further expand to 32-bits. The reference row and column checksum of C matrix is compared against the final accumulated C matrix checksum and similar to previous stages the exact row and column location of the faults is reported and corrected before it is passed on to the non-linear activation function stage. Non-linear functions are either implemented as lookup table with Newton-Raphson methods to refine precision or as cordic-units to reduce lookup table size and on-chip memory. As this stage does not affect the overall area of the chip this is implemented as redundant compute stage with 3X compute logic. Before storing the C panel results row column checksums are computed and stored back in the memory for future layer reference. After activation stage the 32-bit accumulated sum is quantized [8] back to 8-bits as this becomes the next set of features for subsequent layers.

For all the logic units to work in parallel sufficient stage pipelining and buffering need to be handled as shown in Figure 6. Reference checksums must arrive before computing checksums and comparisons. Sufficient timing also has to be allotted for correction once the exact row, column of fault is identified. It takes 64 cycles to read one full matrix of 64x64 using 512bit bus. There are two dedicated bus for read (A & B panel bus) and one dedicated bus for write (C panel bus). The A matrix is fetched in blocks of 64 as [A_1, A_2, A_3 … A_N]. The B matrix is also fetched in blocks of 64 but is kept in double buffered panel as [B_1, B_2, B_3, . B_N]. The C panel output is also broken down into 64x64 chunks but is again double buffered [C_1, C_2, C_3, … C_N] Using the 128-bit checksum bus on the input side, A and B checksums arrive in 32 cycles. A and B error computation and correction before the next 64-cycle window begins where the actual matrix multiplication happens [$C_1 = A_1 \times B_1$]. At the end of the 2nd 64-cycle window C error computation occurs and output is corrected before the C panel accumulation begins in the 3rd 64-cycle window. At this time C reference checksum can arrive, note this is 32-bit checksums as there is data expansion due to multiplication and accumulation. This operation repeats till all accumulation iterations are complete and the final C panel sum is forwarded to non-linear activation function stage. Here results are generated thrice, compared for consistency and corrected for fault if any. Before writing the output, checksums are computed and quantized back to 16-bit for next layer processing. In the steady-state all the stages take deterministic amount of time and executed in parallel to meet real-time performance requirements. The 64-cycle window is a function of the panel size. Proper consideration of panel size and bus width is crucial to accommodate reference checksum fetch to arrive in the background before the computations begin.

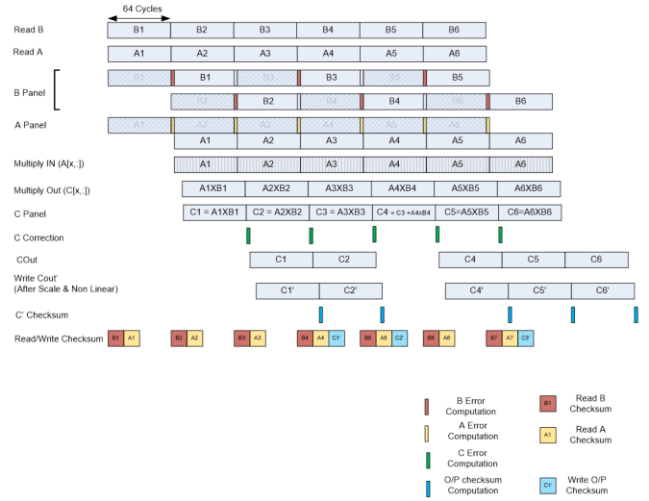

Figure 6. Pipeline sequencing of various stages. Checksum fetch and compute has to run in parallel to actual compute stages, just in time to perform comparison and correction

Simulation and Results

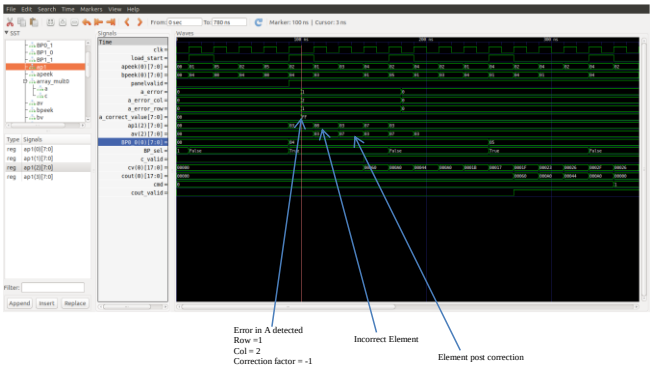

Figure 7. HDL simulated timing diagram which shows error detected in A panel with errors in row 1 and column 2 which is detected in next clock and corrected in the following

The proposed solution is HDL [12] coded and simulated. Simulation shows the ability to detect error in matrix A, B, C along with full performance. High level waveform shown in Figure 7 shows error is detected in A panel and corrected in the following cycle. Table 1 shows a baseline estimate for a typical matrix multiply engine adding to roughly 25.09M gates.

Table I: Baseline area estimate for a typical matrix multiply engine

Module	Gate count	Comments
ECC A & B	18K	
A panel	4K	64 8-bit elements
B panel	521K	2 panels, 4096 8-bit elements
Dot product block	20.16M	64 vector multipliers
Accumulator	128K	64 32-bit adders
C panel	1024K	4096, 32-bit elements
Activation function	48K	
ECC calculation	8K	
Total gate count	21.86M	

Table II lists the gate count for the proposed solution which includes logic for ECC, correction circuitry and additional buses to carry reference checksums. As it adds up, the proposed fault tolerant design adds less than 15% of gate count compared to 3X gate count with a fully redundant design.

Table II: Proposed architecture gate count

Module	Gate count	Comments
Checksum A&B	198K	
A panel	256K	4096, 8-bit elements
B panel	512K	2 panels, 4096 8-bit elements
ECC A panel	48K	Generation, storage & correction
ECC B panel	576K	Generation, storage & correction
A correction	32K	64-bit adder
Dot product block	20.16M	64 vector multipliers
Checksum product	1252K	2 vector multipliers (64 x 8-bit * 16-bit) + storage
A*B correction	96K	64 24-bit adder
Accumulator	128K	64 32-bit adder
C Panel	1024K	4096 32-bit elements
ECC C Panel	192K	Generation, storage & correction
Checksum C	360K	2 * 64, 40-bit adder + storage
C correction	128K	64 32-bit adder
Activation function	140K	3X
Checksum on C'	99K	
Total gate count	25.09M	

Conclusion and future work

In this paper we have discussed novel ways of designing a fault tolerant system with just a marginal increase in area over a baseline design without any memory or compute protections. This is a significant improvement in overall area and cost of the solution over a fully redundant (3X) fault tolerant system. Future work involves exploration of minimizing the 15% overhead of area by not checking fault and correcting it at every stage but delay correction at key stages of just multiplication or accumulation.

References

[1] https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level

[2] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classifcation with deep convolutional neural networks. In: Advances in neural information processing systems, 2012: 1097–1105

[3] M. Mody et al., "High Performance Front Camera ADAS Applications on TI's TDA3X Platform," 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), Bengaluru, India, 2015, pp. 456-463, doi: 10.1109/HiPC.2015.56.

[4] M. Mathew, K. Desappan, P. K. Swami, S. Nagori, and B. M. Gopinath, "Embedded low-power deep learning with tidl," Texas Instrum., Dallas, TX, USA, Tech. Rep. SPRY314, 2018.

[5] Ngiam, Jiquan & Khosla, Aditya & Kim, Mingyu & Nam, Juhan & Lee, Honglak & Ng, Andrew. (2011). Multimodal Deep Learning. Proceedings of the 28th International Conference on Machine Learning, ICML 2011. 689-696.

[6] Kung, "Why systolic architectures?," in Computer, vol. 15, no. 1, pp. 37-46, Jan. 1982, doi: 10.1109/MC.1982.1653825.

[7] E. J. McCluskey, "Built-In Self-Test Techniques," in IEEE Design & Test of Computers, vol. 2, no. 2, pp. 21-28, April 1985, doi: 10.1109/MDT.1985.294856.

[8] R. W. Hamming, "Error detecting and error correcting codes," in The Bell System Technical Journal, vol. 29, no. 2, pp. 147-160, April 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[9] K Desappan, et.al, "CNN Inference: Dynamic and Predictive Quantization", IEEE International Conference on Consumer Electronics, (ICCE) , Berlin, 2018.

[10] Agarap, Abien Fred. (2018). Deep Learning using Rectified Linear Units (ReLU).

[11] Vachhani, Leena & Sridharan, K. & Meher, P.K.. (2009). Efficient CORDIC Algorithms and Architectures for Low Area and High Throughput Implementation. Circuits and Systems II: Express Briefs, IEEE Transactions on. 56. 61 - 65. 10.1109/TCSII.2008.2010169.

[12] https://en.wikipedia.org/wiki/Hardware_description_language

Author Biography

Shyam Jagannathan is an Edge AI architect and Senior Member of Technical Staff (SMTS) at Embedded Processors Group, Texas Instruments. His domains of interest include DSP architecture, SoC architecture, hardware accelerators, deep learning, perception, sensor fusion localization, path planning and overall system optimization He received a master's degree in the field of Signal Processing and Communications from Illinois Institute of Technology, Chicago in 2013

Mihir Mody is SoC Architect lead and Distinguished Member of Technical Staff (DMTS), responsible for roadmap and chip definition for Application Specific MCU business in Texas Instrument (TI). His domains of interest are real time control, image processing, computer vision, deep learning and Video coding. He received his master's in electrical engineering from Indian Institute of Science (IISc) in 2000

Prithvi Shankar is SoC Architect and Senior Member of Technical Staff (SMTS) responsible for chip definition for Application Specific MCU business in Texas Instrument (TI). His domains of interest are VLSI design, IP and SoC architecture and use case modelling.

Jesse Villarreal is a software architect for TI's heterogeneous multicore SoCs and a Senior Member of Technical Staff (SMTS) at Embedded Processors Group, Texas Instruments. He received a master's degree from the University of Texas at Dallas in Computer Engineering and has been with Texas Instruments since 2001. His areas of interest include DSP software optimization, heterogeneous multicore middleware frameworks, vision and imaging hardware accelerators, and overall system software scalability, portability, and optimization

JuneChul Roh is a Senior Systems Architect and a Senior Member of Technical Staff (SMTS) at the Embedded Processors Group, Texas Instruments. His interests include signal processing, deep learning, radar, edge AI, and robotics systems and applications. He received his Ph.D. degree in Communications and Signal Processing from the University of California, San Diego, in 2005

Kumar Desappan is Senior Member of Technical Staff (SMTS) at Texas Instruments (TI) Incorporated. His domains of interest are Machine/Deep learning, image processing and computer vision algorithms with a focus on software solution for edge devices. He received Bachelor of Engineering (BE) from Anna University - Chennai in 2005

Deepak Poddar is software development manager and Senior Member of Technical Staff (SMTS) for Embedded processors business unit in Texas Instrument s(TI). His domains of interest are image processing, computer vision, deep learning and Video coding. He received his bachelor's degree in electrical engineering from National Institute of Technology, Warangal in 2000

Pramod Swami is Distinguished Member of Technical Staff (DMTS) at Processors Business in Texas Instruments (TI) leading the software development for EdgeAI processing. His domains of interest are Embedded systems, Digital Signal Processors, Deep Learning, Computer Vision, Image Processing, and Video coding. He received his Bachelor's degree in Electronics and communication engineering from Malaviya National Institute of Technology (MNIT) Jaipur in 2001