3

Tools and Methodologies for Training, Profiling, and Mapping a Neural Network on a Hardware Target

Alexandre Valentian[1], Simon Narduzzi[2], Muhammad Arsalan[5], Kay Bierzynski[5], Stefano Traferro[4], Preetha Vijayan[4], Amirreza Yousefzadeh[4], Manolis Sifalakis[4], Rene Van Leuken[8], Dylan Muir[7], Rashid Ali[3] Maen Mallah[3], Bijoy Kundu[3], Loreto Mateu[3], and Mario Diaz Nava[6]

[1]CEA, France
[2]CSEM, Switzerland
[3]Fraunhofer IIS, Germany
[4]Imec, The Netherlands
[5]Infineon, Germany
[6]STMicroelectronics, France
[7]SynSense, Switzerland
[8]TU Delft, The Netherlands

Abstract

The European project ANDANTE [1] aims at providing neuro-inspired and/or energy-efficient hardware accelerators for running AI applications at the edge. Given the wealth of applications targeted, with various processing needs and sensors involved, several implementations are pursued in parallel: (1) fully digital or analog-mixed signal; (2) with classical coding or spike coding; (3) leveraging different embedded Non-Volatile Memory (NVM) technologies. However, what do all have all in common? it's the need for adequate tools and methodologies for training and deploying neural network models, considering hardware constraints. This Chapter provides details on what has been developed and used in the frame of the ECSEL ANDANTE

project. Firstly, a neural network must be learnt, considering limited hardware resources, thus exploiting quantization and sparsity for instance. When dealing with Spiking Neural Networks (SNNs), the training phase is even more critical, depending on the neuron model and making use of various strategies (direct training versus conversion). Then, the network must be mapped on the target accelerator, be it a spatially folded or spatially expanded architecture. In the latter case, graph transformation might be needed using a compiler. Finally, Key Performance Indicators (KPIs) must be extracted, underlying the need for a simulator/profiler.

Keywords: Machine learning, deep learning, artificial neural networks, spiking neural networks, optimisation, quantization, pruning, distillation, NN model transformation, sparsity, ANN-to-SNN conversion, accuracy, NN compiler, mapping strategy, simulation, profiling.

3.1 Introduction

3.1.1 Edge Computing Benefices and Challenges

Edge computing is creating new opportunities for Internet of Things (IoT) applications. Through machine learning, objects become intelligent and can process a large amount of information. However, most of this processing today still takes place in the cloud, and it comes at several costs: infrastructure, reliability, security, speed, and energy. Firstly, the infrastructure to process data from heterogeneous devices needs an extensive infrastructure to gather, transform, and store the data and the devices themselves need connectivity and the corresponding energy to send the data. Therefore, having the data stored and analysed at the edge can reduce the infrastructural costs, save energy, and increase globally data processing efficiency. Secondly, for applications that are critical and need high availability (such as pipeline monitoring), a reliable and secure connection is necessary.

Having devices that can decide at the edge can mitigate the risk associated with the loss of connection and prevent data from being accessed by a third party to ensure security and privacy. Moreover, intelligent edge devices are also necessary for applications where decision speed (low latency) is critical, such as autonomous driving, as having data transferred to the cloud is inconceivable: the latency associated with the connection might result in the life or death of people. Finally, the energy associated with the transfer and data processing in the cloud is still enormous: 40% of the energy used in mobile streaming comes from the mobile cellular network. Therefore, in

the context of the climate crisis, edge machine learning can substantially reduce the carbon footprint associated with data processing in the cloud. The reduction of infrastructure costs, reduction of communication bandwidth, improvement of security and privacy, and availability of services are by-products of deploying efficient, intelligent, and cost-effective devices at the edge.

With the wide acceptance of Deep Learning (DL) in the last decade, it has become evident that classic deep learning cannot scale, as performant networks use an enormous amount of energy and memory capacity, and the models are becoming larger as their computational power needs increase. Moreover, Moore's law is ceasing to apply, and we need new computational paradigms to increase computational performance with a reduced energy budget. With the evolution of IoT devices, deep learning models are now deployed at the edge, allowing local real-time decision-making, efficient pre-processing, and privacy-preserving applications. Optimizations have been developed in the past few years to allow the deployment of these networks within restricted resource environments; quantization, pruning, distillation, are some of them, which are either applied during training or post-training of the neural network. While these techniques offer a partial solution for the deployment on edge devices, a lot of engineering is still required to design models that fit within the constraint of the hardware. An alternative emerging machine learning technology to reduce energy relies on SNNs, which are structures imitating the neurons in the brain. Their computational efficiency is thought to be due to the coding style of the biological neurons, which communicate using electrical discharges, called spikes, that travel from one neuron to the other using synaptic connections.

While industry leaders Intel, ARM, Google, and NVIDIA are developing systems targeting large-scale computation based on Graphics Processing Units (GPUs) or specialised AI processors for generic AI applications in the cloud, a parallel branch targets low-power applications at the edge, with algorithmic solutions that will only be efficient if they can run on suitable hardware solutions. Currently, much effort is put into developing low-power accelerators for artificial neural networks, and to some extent, spiking neural network. Academia is also putting effort into the development of technologies targeting edge processing: the recent development of memristors and Ferroelectric Field-Effect Transistor (FeFET) technologies herald a new era of ultra-low-power hardware to accelerate neural networks [21] [22].

While most accelerators target generic applications, there are still many limitations on the hardware that make them suboptimal for specific tasks:

limitation in speed, memory size, supported operations (spiking or digital), or energy consumption. This wide choice of embedded systems makes it challenging to identify the relevant hardware suitable for a particular application. This major challenge restricts the adoption and dissemination of ultra-low-power applications, as many efforts are put into studying and researching the most suitable device.

3.1.2 Artificial Neural Networks (ANNs) and Spiking Neural Networks (SNNs)

In biology, neurons communicate through current inputs called action potentials, or spikes. When a neuron receives input stimuli (spikes) from other neurons, they depolarize the neuron cell membrane by changing the concentration of ions inside and outside of the cell membrane, creating a potential. The strength of the depolarization depends on the strength of the synaptic connection between the pre- and post-synaptic neurons. The succession of depolarizations events leads to an increase of the membrane potential. If the cell membrane potential increases to a precise threshold voltage, it triggers a cascade effect leading to the emission of a spike.

In 1958, Frank Rosenblatt created the first model of a neuron generating binary decisions, simulating the emission of a spike, or not. The perceptron was a single neuron model performing computation using multiple weighted input values, simulating the strength of the synaptic connections, using a weight matrix. When the weighted sum of input reached a certain value, the neuron output switched. Current deep learning relies on variants of this algorithm, by creating stacked structures (layers) of neurons that combine and transform the information in a non-linear manner, resulting in impressive performance in a wide variety of tasks.

Deep learning algorithms can be accelerated on dedicated hardware to provide low-power solutions for edge applications. ASICs for deep learning inference accelerators offer better area and energy efficiency than GPUs, FPGAs or CPUs but at the cost of less flexibility [1]. Since ANNs perform multiply and accumulate (MAC) operations, the hardware pursues to acceleration such operations by parallelizing them. To overcome the von Neumann bottleneck, ASIC architectures based on analog in-memory computing with crossbar arrays to perform the MAC operation are pointed out as a relevant solution when it comes to low latency and high energy efficiency. Such inference accelerators have on-chip memory buffers as well as processing elements where the weights are stored individually to avoid data movement during inference.

Although inspired by the biological nervous systems, ANNs are yet unable to capture the sophisticated neurocomputational features of biological neurons. To bridge this gap, DL community has come up with a third generation of ANNs known as SNNs. SNNs are more closely mimicking biological neural networks than artificial neural networks that are rate-based. This type of neuron is represented by a membrane state and therefore incorporates the concept of time. Spiking neural networks, in contrast to artificial ones, only send "spikes" and not digital values. However, they can represent values using spike trains, which rate can be as equivalent to values processed by artificial deep neural networks.

Research on spiking neural networks is still on-going. The recent development of neuromorphic hardware platforms has allowed simulation of large-scale brain models. However, how to perform deep learning using these types of neurons is still unclear. In particular, a lot of different types of spiking neurons exist, and no standard has been agreed on yet. This wide variety of neurons must also be considered by designers of neuromorphic hardware, so that researchers can assess the suitability of models.

The neurons in SNN are described on different abstraction levels starting from the most realistic and complex model, Hodgkin-Huxley (HH) model, to the leaky integrate-and-fire (LIF) model which is the simplest and most computationally efficient model bearing the neurocomputational properties [23]. LIF introduces a leaky term to the integrate-and-fire (IF) model that causes neuron potential decay over time making it more biologically plausible.

With the advancement of research on spiking neural networks, academia and industry have developed accelerators and processors specialized in supporting this type of algorithm. A few research institutes and companies develop large-scale hardware solutions to simulate spiking neural networks, like SpiNNaker, IBM TrueNorth, and Intel Loihi and Loihi 2. While reasonably accurate at simulating large-scale brain dynamics, these processors do not target ultra-low-power edge applications and still use a considerable amount of energy. As their primary purpose was to simulate SNN, the processing happening in these accelerators is unsuitable for common industrial applications, as developed nowadays using deep learning. Research is still actively investigating suitable event-based device for industrial applications, and now we observe the emergence of new hardware accelerator relying on binary events computed in a synchronous manner, meeting halfway between the pure asynchronous SNNs and the synchronous processing of ANNs.

In the ANDANTE project, this type of efficient hardware neuromorphic accelerators is being addressed based on new embedded memory technologies such as PCM, OxRAM, MRAM and FeFET, novel ANN/SNN architectures combining analog digital mixed-signal designs, which call for dedicated tools and methodologies.

The rest of this Chapter is organized as follows: Section 1.2 provides the state-of-the-art in neural network training, exploiting quantization, sparsity and showing different strategies for training spiking neural networks. Section 1.3 presents further refinements to sparsity, to exploit temporal sparsity (in addition to the weight and activation ones) by adding a new layer called Temporal Delta Layer. Section 1.4 describes how to map a neural network onto a spatially expanded, in memory computing-based architecture: in such a case, the neural network weights must be adequately clustered or duplicated on the various NVM arrays. Finally, Section 1.5 shows the mapping of spiking neural networks on a hardware target implementing LIF neurons with recurrent connections. Finally, Section 1.6 gives clues on why it is important to profile a neural network topology.

3.2 State-of-the-art of key aspects of Neural Networks

3.2.1 ANN and SNN Hardware Aware Design

Hardware-aware design of artificial and spiking neural networks is still a multistep process. Since no generic design and simulation tools are available for custom neuromorphic hardware platforms, it is still required to deploy the model on the physical devices to obtain the key performance indicators of the application. As shown in Figure 3.1, the optimization of a neural network for a specific edge device is an iterative process involving hardware/software co-design. The first iterative cycle is the development of an accurate model solving the task to which it is designed, and the second cycle consists in the embedding and evaluation of the model. The model can then be further optimized towards the optimization of edge KPIs, necessitating a new training iteration phase followed by deployment. The process can be automatized using automated search procedures like Network Architecture Search (NAS), which have demonstrated to be suitable solutions for the design of a model respecting the constraint of their end-deployment platform [77]. However, this framework still contains major challenges. Indeed, the deployment on a device is often complicated and requires a manual adaptation of the model to allow the neural network to run on the device. Moreover, some platforms have a specific instruction set or a variable data representation (float or integer),

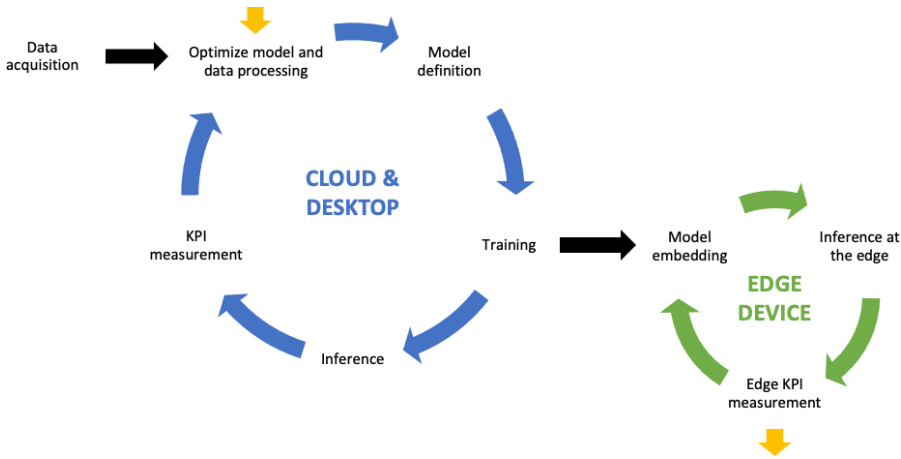

Figure 3.1 Networks to hardware workflow.

requiring a quantization step either during training (defined at the creation of the model) or after training, usually impacting the overall performance of the network. Finally, platforms are often behind the latest software developments in the creation of layers in neural networks, which makes some architectures impossible to deploy due to the existence of unsupported layers.

Regarding the automatic search of architectures for a certain platform, the computational cost is still very intensive and usually must be replicated for each new platform, despite recent improvements in this direction [69]. Therefore, flows and techniques have been developed to design efficient neural networks for neuromorphic hardware platforms. Some of them are described in the next paragraphs.

3.2.2 Sparsity

Reducing energy consumption is a critical point for neural network models running on edge devices. In this regard, reducing the number of MAC operations of DNNs running on edge hardware accelerators will reduce the energy consumption during inference. Optimizations have been developed in the past few years to allow the deployment of these networks within restricted resource environments; quantization [2], pruning [3], distillation [4], are some of them, which are applied either during training or post-training of the neural network. Great emphasis is also put on the development of efficient accelerators, that reach competitive performance compared to

CPUs and GPUs. Recent hardware accelerators include optimization techniques such as computational reduction by zero-skipping [5–7], that skip zero weight computation in and are therefore optimized for very sparse neural networks.

Efforts have been made toward the sparsification of deep neural networks to reduce the memory footprint of the models deployed at the edge. Pruning is a method used to achieve weight [12] and feature map [13–15] sparsification to remove redundant information and subsequently reduce network computations. In SNNs, spikes and synaptic computation reduction are mostly exploited through temporal and spatial sparsity. Temporal sparsity of SNNs have inspired training techniques in deep learning [16, 17], targeting time-series applications. Recently, regularization techniques have been applied to SNN training [18, 19] to increase spatial sparsity, and during BP-trained DNNs training prior to SNN conversion [11, 20, 79].

3.2.3 ANN-to-SNN Conversion

Spiking neural networks can potentially save much more energy than continuous-valued Artificial Neural Networks due to their sparse nature and event-driven computations. While SNNs may provide a large panel of advantages, their training is still complicated, as the current hardware and training algorithms are not suitable to train SNN in an asynchronous manner. Therefore, one common technique to create performant SNNs is to convert them from a previously trained ANN.

Early attempts to convert ANN to SSN comprise the work of [70] where neurons of a Convolutional Neural Network were transformed to leaky-integrate and fire (LIF) neurons with refractory periods. A similar technique [71] used a weight normalization scheme in an ANN to regulate the firing rate of the converted SNN. Another work [72] developed a conversion method using spiking neurons that adapt their firing threshold to reduce the number of spikes needed to encode information.

One largely used technique of conversion of ANN to SNN has been developed by Rueckauer [10]. It is based on scaling of the weights of the pretrained SNN such that the firing rate of the neurons match the activation values of the ANN. While this technique supports a wide range of layers, it requires a long simulation time for the model to reach competitive accuracy. Recent methods [73–75] adjust the threshold values of the neurons to reduce the inference latency.

3.2.4 Surrogate Gradient Descent

Spiking neuron models commonly incorporate highly non-linear transfer functions, such as the Heaviside function, to map from internal state variables to binary output events.

$$S\left(V_{mem}\right) = H\left(V_{mem} - V_{th}\right) \tag{3.1}$$

These functions often have poorly behaved or undefined derivatives. In the example here $dS/dV_{mem} = 0$ everywhere. When used in conjunction with gradient-based optimisation methods such as error backpropagation [9], these poorly behaved derivatives propagate to cause the gradients of parameters to be not informative. Standard gradient-based training techniques cannot therefore be directly applied to SNNs.

One method to work around this limitation is to define a *surrogate gradient* for the SNN transfer function. In this approach the derivative of the transfer function is defined using an auxiliary "surrogate" function, ideally with similar behaviour to the true transfer function, but with better-behaved derivatives. For example, instead of the non-linear Heaviside function, a ReLU function can be used as an approximation for computing the gradient in the backwards pass.

$$\hat{S}\left(V_{mem}\right) = \max\left(V_{mem} - V_{th}, 0\right) \tag{3.2}$$

$$\frac{dS}{dV_{mem}} \equiv \frac{d\hat{S}}{dV_{mem}} = V_{mem} > V_{th} \tag{3.3}$$

This method permits SNNs to be trained using gradient-based optimisation algorithms such as SGD [78–80] and Adam [81]. Recently this approach has been used to integrate SNNs with industry-standard automatic differentiation libraries such as PyTorch and Jax, to permit training of deep SNNs [82]. In this way not only the weights of a network can be optimised, but in addition all the auxiliary parameters of an SNN such as time constants and thresholds [82].

3.2.5 Neural Engineering Object (Nengo) Simulator

The Neural Engineering Object (Nengo) is a neural network simulation tool for large-scale neural systems with applications in cognitive science, psychology, AI, and neuroscience [24]. Nengo offers NengoDL, a deep learning simulator, which enables for easy integration of the TensorFlow library and access to advanced features such as convolution connections. Using a neural

engineering framework NEF. Nengo designs neural network models for application in machine learning and deep learning such as inductive reasoning, gesture sensing [25], action selection, speech production [26, 83] and image classification [84], etc.

NengoDL uses NEF for building neuron models for building biologically plausible neural networks. NEF provides the principles of representation, transformation, and dynamics to construct a neural model. The NEF encodes the incoming time varying input data of real numbers and based on the input data; a specific amount of current is injected into a single neuron model. This current causes the neuron to spike and the spiking behaviour is controlled by the tuning the curve of the neuron models. The tuning curve is determined by the bias, gain of the neuron and the encoding weights. In the decoding stage, an exponentially decaying filter is applied to the spike train resulting in a spike generating postsynaptic current [24].

The strength of the postsynaptic current is defined by its amplitude which is affected by various factors. The NEF summarizes these factors in the form of a connection weight matrix representing the strength of the connection between two neural populations. These matrices can be factorized into smaller matrices allowing to efficiently run large-scale neural models on low commodity hardware [24].

An information is represented by a Nengo ensemble, and a connection defines how the information is transformed. Nengo uses an object model to translate the ensembles and their connections into a network of interconnected neurons. In this way, it acts as a "neural compiler", converting high-level functional models to low-level models. Nengo defines six core objects as an object model: 1) ensemble, 2) node for non-neural information such as sensory inputs, 3) connection, 4) probe for data collection during simulation, 5) network for interconnected nodes and ensembles, and 6) model. Because of the separation of model construction and simulation, Nengo models can be used on a variety of simulators. [24].

In addition to the biological plausible neurons, nengoDL allows to use rate-based neurons such as LIFRate, Rectified Linear, Sigmoid and Tanh by converting them to their spiking version using wrappers that take some function and return an instantaneous firing rate. These wrappers are:

1) **Regular Spiking**: takes the instantaneous firing rate and integrates it multiplied by a timestep.
2) **Poisson Spiking**: Given an instantaneous rate, this wrapper draws a sample from a Poisson distribution. The value of the distribution is this instantaneous firing rate.

3) **Stochastic Spiking**: is kind of a mix between the two, and the difference mostly shows up when neurons can spike more than once per timestep.

In Figure 3.2, some conversion examples are illustrated. These neurons are created by employing Regular Spiking to convert rate-based neurons to their spiking counterparts. For example, Figure 3.2(a) is created $\tau_{ref} = 0.0025$ indicating that the firing will saturate at 400 Hz. The neuron begins in a blank state (i.e., no input current, no membrane current, etc.), implying that the neurons are doing nothing when the simulations begin, and it takes a few time steps for the neuron to get going. The curve becomes a little noisy around the middle because the neuron has modest firing rates and so few spikes in that area. Moreover, it can be seen that the neuron is showing two kinds of spikes, positive and negative. Because this type of spiking behaviour isn't biologically reasonable, it won't operate on most neuromorphic technology. Similarly, Figures 3.2(b), (c), and (d) represent the spiking version of Sigmoid, Rectified Linear and LIFRate based neuron. It should be noted that the curve's slope is determined by the neuron's gain. The gain of the neurons has been modified in these cases to produce less noisy curves.

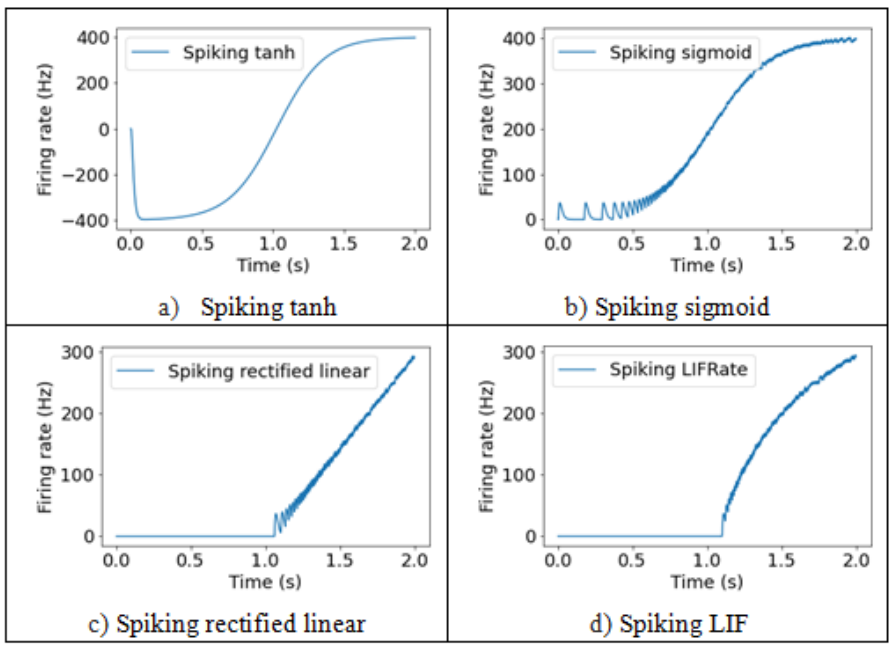

Figure 3.2 Spiking neuron models.

3.3 NN Transformation: Temporal Delta Layer

This Section focuses on a transformation applicable to DNNs which generates temporal activation sparsity during training and exploit it during inference.

The energy consumed by running DNNs on hardware accelerators is dominated by the number of memory read/writes and multiply-accumulate (MAC) operations. As a potential solution, the role of activation sparsity in efficient DNN inference is proposed. i.e., as the predominant operation in DNNs is matrix-vector multiplication of weights with activations, skipping operations and memory fetches where (at least) one of them is zero can make inference more energy efficient.

In this Section, a new DNN layer (called temporal delta layer) whose primary objective is to induce temporal activation sparsity during training is presented. The temporal delta layer promotes activation sparsity by performing delta operation through activation quantization and l_1 norm-based penalty to the cost function. During inference, the resulting model acts as a conventional quantized DNN with high temporal activation sparsity.

3.3.1 Temporal Delta Layer: Training Towards Brain Inspired Temporal Sparsity for Energy Efficient Deep Neural Networks

DNNs have lately managed to successfully analyses video data to perform action recognition [27], object tracking [28], object detection [29], etc., with human-like accuracy and robustness. Unfortunately, the high accuracy of DNNs comes with high compute and memory costs, resulting in high energy consumption. This makes them infeasible for always-on edge devices.

Over the years, techniques like network pruning, quantization, regularization, and knowledge distillation [30–32] have helped in reducing the model size footprint resulting in overall lesser computation and memory consumption. Noticeably, sparsity is an underlying feature in all the solutions. This is notable, as sparse tensors provide the potential to skip computations that involve multiplication with zeroes. Also, they are easier to store and access in memory. Structural sparsity (of weights) and spatial sparsity (of activations) are well-researched topics in DNN literature [33]. However, temporal activation sparsity is comparatively less explored in the context of DNN, although it is a popular concept in neuromorphic computing.

The concept of change or delta-based processing is taken from the human retina to the training and inference phases of deep neural networks [34]. DNN inference which processes each frame separately with no regard to the

temporal correlation is dense and obscenely wasteful. Whereas processing only the changes in the network can lead to zero-skipping in sparse tensor operations reducing redundant operations and memory accesses.

Therefore, the proposed methodology in this work induces temporal sparsity to potentially any DNN, by means of a new layer (called Temporal Delta Layer), which can be introduced in a DNN at any phase (training, refinement, or inference only). This new layer can be integrated into an existing architecture by placing it after all or some of the ReLU activation layers as deemed computationally beneficial (see Figure 3.3).

The inclusion of this layer does not require any change to the preceding and following layers. Moreover, during the training phase, the new layer adds a novel sparsity penalty to the overall cost function of the DNN. This l_1 norm-based penalty minimizes the activation density of the delta maps (i.e., temporal difference between two consecutive feature maps). Apart from that, two activation quantization methods, namely fixed-point quantization (FXP) and learned step-size quantization (LSQ), are also compared in conjunction with the new layer.

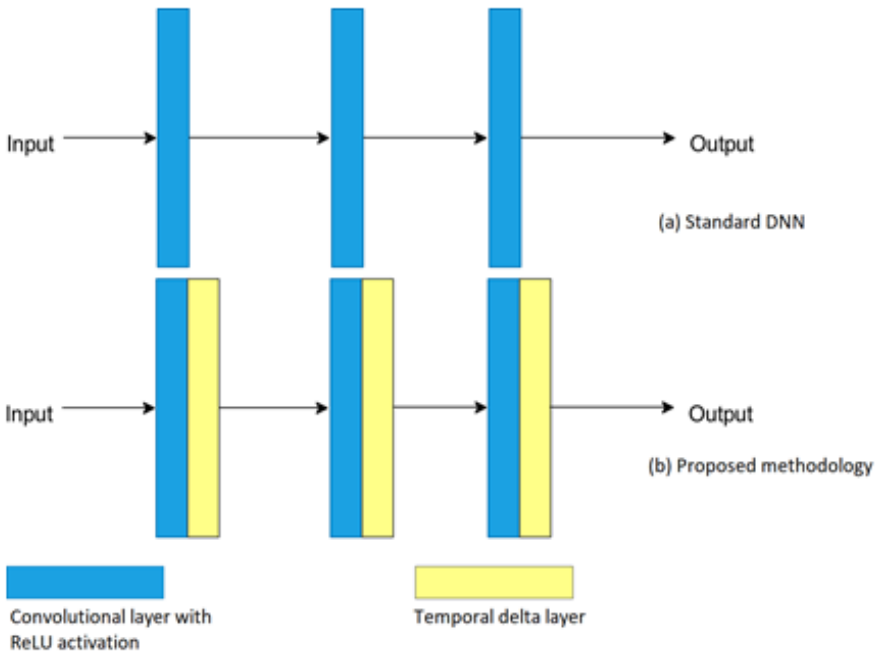

Figure 3.3 (a) Standard DNN and (b) DNN with temporal delta layer.

The inclusion of this layer does not require any change to the preceding and following layers. Moreover, during the training phase, the new layer adds a novel sparsity penalty to the overall cost function of the DNN. This l_1 norm-based penalty minimizes the activation density of the delta maps (i.e., temporal difference between two consecutive feature maps). Apart from that, two activation quantization methods, namely fixed-point quantization (FXP) and learned step-size quantization (LSQ), are also compared in conjunction with the new layer.

3.3.2 Related Works

Although DNNs are in essence bio-inspired, they have not been able to find the balance between power consumption and accuracy yet, especially while dealing with computationally heavy streaming signals. On the other hand, the brain's neocortex handles complex tasks like sensory perception, planning, attention, and motor control while consuming less than 20 W [35]. Scalable architecture, in-memory computation, parallel processing, communication using spikes, low precision computation, sparse distributed representation, asynchronous execution, and fault tolerance are some of the characteristics of the biological neural networks that can be leveraged to bridge the energy consumption gap between the brain and DNNs [36]. Among these, the proposed methodology focuses on the viability of using sparsity within DNNs to achieve energy efficiency. During a matrix-vector multiplication between a weight matrix and an activation vector, zero elements in the tensor can be skipped leading to computational as well as memory access reduction (see Figure 3.4).

There are broadly two types of sparsity available in DNNs: weight sparsity (related to the interconnect between neurons) and activation sparsity (related to the number of neurons). Furthermore, activation sparsity can be categorized into spatial and temporal sparsity, which exploits the spatial and temporal correlation within the activations, respectively, [38]. Unlike weight and spatial sparsity [39–44], exploiting the temporal redundancy of DNNs while processing streaming data to reduce energy consumption is a relatively less explored idea. Exploiting temporal sparsity translates to skipping recalculation of a function when its input remains unchanged since the last update.

One of the methods to exploit temporal sparsity is to use the compressed representation (like H.264, MPEG-4, etc.) of videos at the input stage itself. These compression techniques only retain a few key-frames completely and

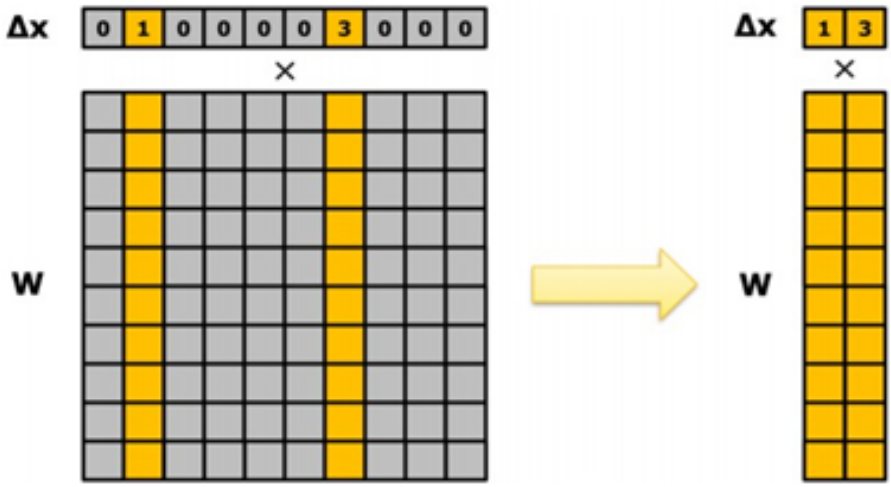

Figure 3.4 Sparsity in Δx can save multiplications between Δx and columns of W that correspond to zero [37].

reconstruct others using motion vectors and residual error, thus using temporal redundancy [45, 46]. Another path includes finding a neuron model which is somewhere in between "frame-based DNN" and "event-based spiking neural networks". This Section describes an attempt in the direction. A similar work, CBInfer [7] proposes replacing all spatial convolution layers in a network with change-based temporal convolution layers (or CBconv layers). In this, a signal change is propagated forward only when a certain threshold is exceeded. Likewise, [48] tapped into temporal sparsity by introducing Sigma-Delta Networks, where neurons in one layer communicated with neurons in the next layer through discretized delta activations. An issue when it comes to CBInfer is the potential error accumulation over time as the method is threshold-based. If the neuron states are not reset periodically, this threshold can cause drift in the approximation of the activation signal and degrade the accuracy. Whereas sigma-delta scheme experiments on smaller datasets like temporal MNIST, which might not be a reliable confirmation of the method's effectiveness.

3.3.3 Methodology

In video-based applications, traditional deep neural networks rely on frame-based processing. That is, each frame is processed entirely through all the layers of the model. However, there is very little change in going from one

frame to the next through time, which is called temporal locality. Therefore, it is wasteful to perform computations to extract the features of the non-changing parts of the individual frame. Taking that concept deeper into the network, if feature maps of two consecutive frames are inspected after every activation layer throughout the model, this temporal overlap can be observed. Therefore, we postulate that temporal sparsity can be significantly increased by focusing the inference of the model only on the changing pixels of the feature maps (or deltas).

3.3.3.1 Delta inference

A new layer is introduced that calculates the delta (or difference) between two temporally consecutive feature maps and quantifies the degree of these changes at only relevant locations in the frame. Since zero changes are not propagated through the layer, the role of this layer may be perceived as an "analog event propagation". It is considered an "analog event" as it is not the presence of change, but the magnitude of change that is propagated through. To better understand it mathematically, in a standard DNN layer, the output activation is related to its weights and input vector through Equations (3.4) and (3.5).

$$Y_t = WX_t + B \tag{3.4}$$

$$Z_t = \sigma(Y_t) \tag{3.5}$$

where W and B represent the weights and bias parameters, Xt represents the input vector, and Yt represents the transitional state. Then, Zt is the output vector which is the result of s(.) - a non-linear activation function. t indicates that the tensor has a temporal dimension. However, in the temporal delta layer, weight-input multiplication transforms into,

$$\Delta Y_t = W\Delta X_t = W(X_t - X_{t-1}) \tag{3.6}$$

$$\begin{aligned} Yt &= \Delta Yt + Yt - 1 \\ &= W(X_t - X_{t-1}) + W(X_{t-1} - X_{t-2}) + \cdots + Y_0, \;\; where \;\; Y_0 = B \\ &= WX_t + B, \end{aligned} \tag{3.7}$$

$$\Delta Z_t = Z_t - Z_{t-1} = \sigma(Y_t) - \sigma(Y_{t-1}), \;\; where \;\; \sigma(Y_0) = 0 \tag{3.8}$$

In Equation (3.4), instead of using Xt directly, only changes or ΔXt are multiplied with W. Using the resulting ΔYt, the corresponding Yt can be recursively calculated with Equation (3.5), where Yt-1 is the transitional state obtained from the previous calculation. Equation (3.8) is the final delta activation output that is passed onto the next layer.

Another notable difference between the standard DNN layer and the proposed layer is the role of bias. In delta-based inference, bias is only used as an initialization for the transitional state, Y0 in Equation (1.4). However, since bias tensors do not change over time, their temporal difference is zero and is removed from Equation (3.6).

Now, as the input video is considered temporally correlated, the expectation is that ΔXt and by association ΔZt are also temporally sparse. In essence, the temporal sparsity between consecutive feature maps is cast on the spatial sparsity of the delta map that is propagated. Additionally, Yt in Equations (3.4) and (3.7) are always equal. This indicates that if the input is the same, both standard DNN and temporal delta layer based DNN provide the same result at any time step.

3.3.3.2 Activation quantization to induce sparsity

There is temporal redundancy evident in feature maps of two consecutive frames. However, if looked closely, it can be observed that these feature maps are similar but not identical as shown in Figure 3.5(a) and (b). Therefore, if two such consecutive feature maps are subtracted, the resulting delta map has many near zero values, thus restricting the potential increase in temporal sparsity, Figure 3.5(c). This is mainly due to the higher precision available in the floating-point representation (FP32) of the activations. For example, in IEEE 754 representation, a single precision 32-bit floating point number has 1 bit for sign, 8 bits for the exponent and 23 bits for the significant. It not only leads to a very high dynamic range, but also increases the resolution or precision for numbers close to 0. The number nearest to 0 is about $\pm$1.4 x 10$-$45. Therefore, due to high resolution, two similar floating-point values have difficulty going to absolute zero when subtracted. A plausible solution to decrease the precision of the activations is to use quantization.

A post-training quantization method (fixed point quantization [49]) and a quantization aware training method (learnable step size quantization [50]) are considered for comparison as a temporal sparsity facilitator for the new layer.

3.3.3.3 Fixed point quantization

In this method, the floating-point numbers are quantized to integer or fixed-point representation [49]. Unlike floating point, in fixed point representation, the integer and the fractional part have fixed length. This limits both range and precision. That is, if more bits are used to represent the integer part, it subsequently decreases the precision and vice versa.

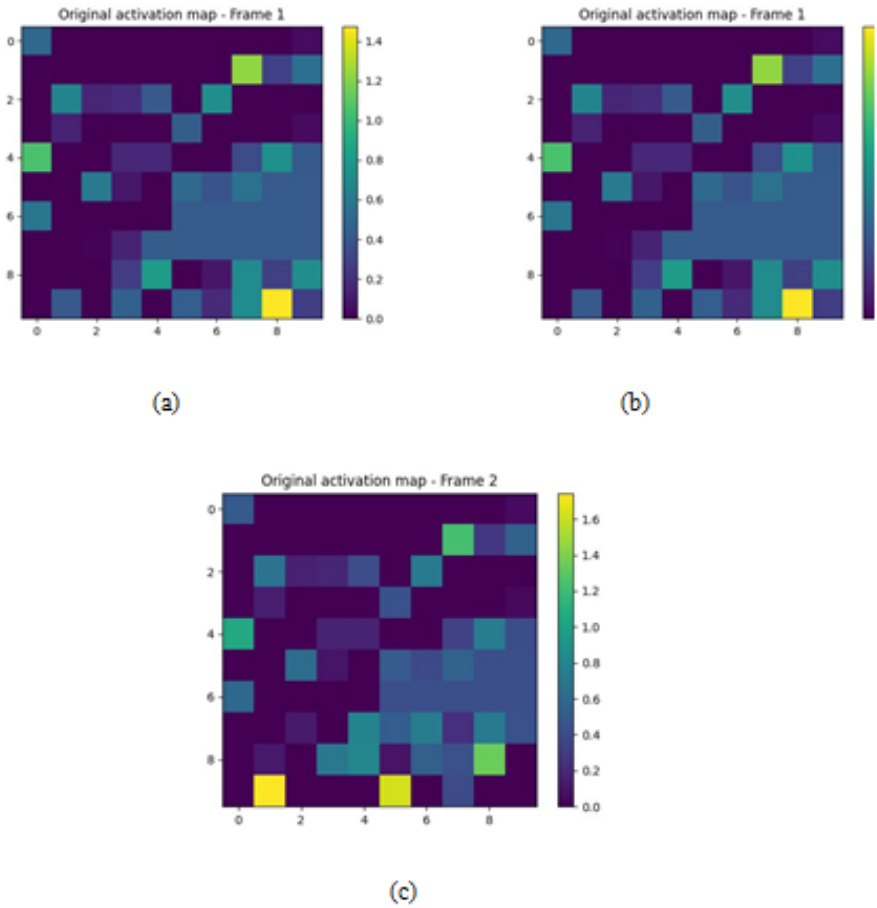

Figure 3.5 Demonstration of two consecutive activation maps leading to near zero deltas.

Method: firstly, a bit-width is defined to which the 32-bit floating parameter is to be quantized, BW. Then, the number of bits required to represent the unsigned integer part of the parameter (x) is calculated as shown in Equation (3.9).

$$I = 1 + \lfloor log_2(max|x|) \rfloor \;\; 1 < i < N \tag{3.9}$$

A positive value of I means that I bits are required to represent the absolute value of the integer part, while a negative value of I means that the fractional part has I leading unused bits. Now, it is known that 1 bit is for

sign, so the number of fractional bits, F, is given by Equation (3.10).

$$\mathrm{F} = \mathrm{BW} - \mathrm{I} - 1 \tag{3.10}$$

Considering the parameters, BW - bit-width, F - fractional bits, I - integer bits, and S - sign bit, Equation (3.11) maps the floating-point parameter x to the fixed point by,

$$Qx = \frac{C(R(x \cdot 2^F), -t, t)}{2^F} \tag{3.11}$$

where *R*(.) is the round function, *C*(*x,a,b*) is the clipping function, and t is defined as,

$$t = \begin{cases} 2^{BW-S}, & BW > 1 \\ 0 & BW \leq 1 \end{cases}$$

The fixed-point quantization, as shown above, is a straightforward mapping scheme and is easy to be included in the model training process during the forward pass before the actual delta calculation. However, it poses a limitation to the extent of quantization possible without sacrificing accuracy. Typically, an 8-bit quantization can sustain floating point accuracy with this method, but if the bit-width goes below 8 bits, the accuracy starts to deteriorate significantly. This is because, unlike weights, activations are dynamic and activation patterns change from input to input making them more sensitive to harsh quantization [51]. Also, quantizing the layers of a network to the same bit-width can mean that the inter-channel behaviour of the feature maps is not captured properly. Since the number of fractional bits is usually selected depending on the maximum activation value in a layer, this type of quantization tends to cause excessive information loss in channels with a smaller range.

3.3.3.4 Learned step-size quantization

Quantization aware training is the most logical solution to the drawback as it can potentially recover the accuracy in low bit tasks given enough time to train. Therefore, a symmetric uniform quantization scheme is considered called Learned Step size Quantization (LSQ). This method considers the quantizer itself as a trainable parameter which is trying to minimize the task loss using backpropagation and stochastic gradient descent. This serves two purposes: (a) step size, which is the width of quantization bins, gets to be adaptive through the training according to the activation distribution. It is vital to find an optimum step size because, as shown in Figure 3.6, if the step size is too small or too large, it can lead to the quantized data being a poor

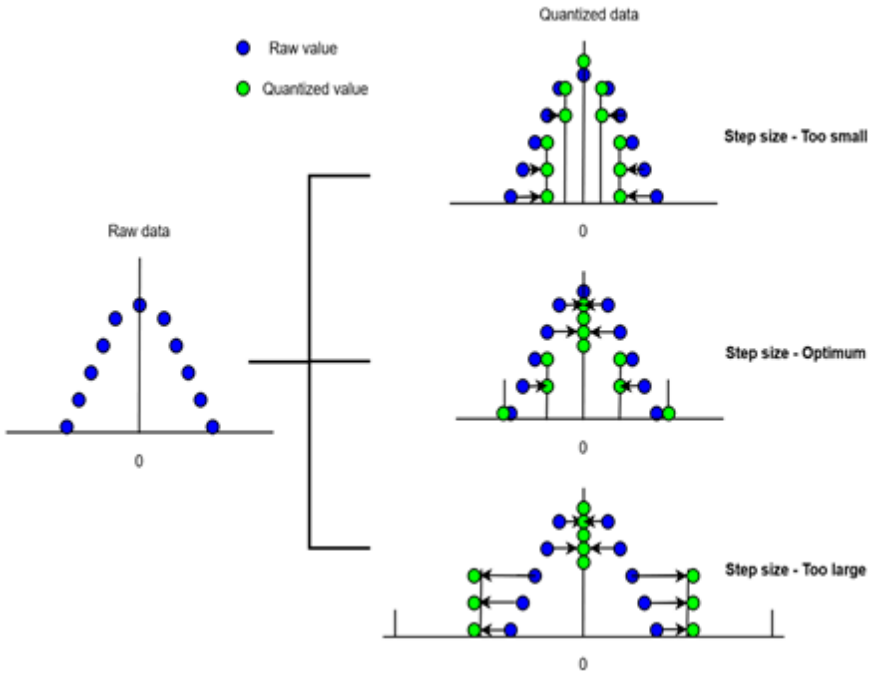

Figure 3.6 Importance of step size in quantization: on the right side, in all three cases, the data is quantized to five bins with different uniform step sizes, but without optimum step size value, the quantization can alter the range and resolution of the original.

representation of the raw data. (b) as the step size is a model parameter, it is also directly seeking to improve the metric of interest, i.e., accuracy.

Method: given: x - the parameter to be quantized, s - step size, QN and QP - number of negative and positive quantization levels respectively, and q(x;s) is the quantized representation with the same scale as x,

$$q(x;s) = \begin{cases} \left\lfloor \frac{x}{s} \right\rceil \cdot s & if -Q_N \leq \frac{x}{s} \leq Q_p \\ -Q_{N,s} & \frac{x}{s} \leq -Q_N \\ -Q_{P,s} & \frac{x}{s} \geq -Q_P \end{cases} \tag{3.12}$$

where $\lfloor a \rceil$ rounds the value to the nearest integer. Considering the number of bits, b, to which the data is to be quantized, $Q_N = 0$ for unsigned and $Q_N = 2^{b-1}$ for signed data. Similarly, $Q_P = 2^{b-1}$ for unsigned and $2^{b-1} - 1$ for signed data.

The original LSQ method is slightly modified to remove the clipping function from the equations as (a) the bit-width, b, required to calculate QN and QP is not known. This is because the bit-width is not pre-defined and is determined using the activation statistics of each layer while training which leads to a mixed precision model, which is more advantageous, and (b) clipping leads to accuracy drop as it alters the range of the activation. That is, if activations are clipped during training, there could be a significant difference between the real-valued activation value and the quantized activation value, which in turn affects the gradient calculations and, therefore, the SGD optimization.

Thus, in temporal delta layer, the forward pass of the quantization includes only scaling, rounding and de-scaling and can be mathematically expressed as,

$$q(x;s) = \left\lfloor \frac{x}{s} \right\rceil \cdot s \tag{3.13}$$

The gradient of the Equation (1.10) for backpropagation is given by Equation (3.14.)

$$\nabla_s q(x,s) = \left\lfloor \frac{x}{s} \right\rceil - \frac{x}{s} \tag{3.14}$$

3.3.3.5 Sparsity penalty

The quantized delta map, created using the above-mentioned methods, has a fair number of absolute zeroes (or sparsity) available. However, like the biological brain, learning can help in increasing this sparsity further. The inspiration for this came from an elegant set of experiments performed by Y. Yu et al. [52]. The experiment showed a particular 30 second video to rodent specimens and tracked their activation density during each presentation. It was found that activation density decreased as the number of trials increased, i.e., as the learning increased, the active neurons required for inference decreased. Adapting the said concept to this work, a l_1 norm-based constraint is introduced to the loss function. This is termed as the sparsity penalty. Therefore, the new cost function can be mathematically expressed as cost function = task loss + sparsity penalty, i.e,

$$\begin{aligned} &Cost\ function \\ &= Task\ loss + \lambda \left(\frac{l_1\ norm\ of\ active\ neurons\ in\ delta\ map}{total\ number\ of\ neurons\ in\ delta\ map} \right) \end{aligned} \tag{3.15}$$

where task loss minimizes the error between the true value and the predicted value and, sparsity penalty minimizes the overall temporal activation density.

The λ mentioned in Equation (1.12) refers to the penalty co-efficient of the cost function. If λ is too small, the sparsity penalty takes little effect and model accuracy is given more priority and if λ is too large, sparsity becomes the priority leading to very sparse models but with unacceptable accuracy. The key is to find the balance between task loss and sparsity penalty.

3.3.3.6 Proposed algorithms

Putting it all together, two algorithms are presented. One uses delta calculations and sparsity penalty concepts with fixed point quantization, and the other uses modified learned step size quantization. The flow charts of the methodology are given in Figures 3.7 and 3.8.

3.3.4 Experiments and Results

The proposed methodology is analysed to study how it helps to achieve the desired temporal sparsity and accuracy.

3.3.4.1 Baseline

For baseline, the two-stream architecture [53] was used with ResNet50 as the feature extractor on both spatial and temporal streams. The dataset used was UCF101, which is a widely used human action recognition dataset of 'in-the-wild' action videos, having 101 action categories [54]. The spatial stream used single-frame RGB images of size (224, 224, 3) as the input, while the temporal stream used stacks of 10 RGB difference frames of size (224, 224, 10×3) as the input. Also, both these inputs were time distributed to apply the same layer to multiple frames simultaneously and produce output that has time as the fourth dimension. Both the streams were initialized with pre-trained ImageNet weights and fine-tuned with an SGD optimizer.

Under the above-mentioned setup, spatial and temporal streams achieved an accuracy of 75% and 70%, respectively. Then, both streams were average fused to achieve a final classification accuracy of 82%. Also, in this scenario, both streams were found to have an activation sparsity of about 47%.

3.3.4.2 Experiments

Scenario 1: The setup consecutively places the fixed-point based quantization layer and temporal delta layer after every activation layer in the network. The temporal delta layer here also includes a l_1 norm-based penalty. Fixed point quantization, in this setup, is used to decrease the precision of input activation maps. Both techniques promote temporal sparsity

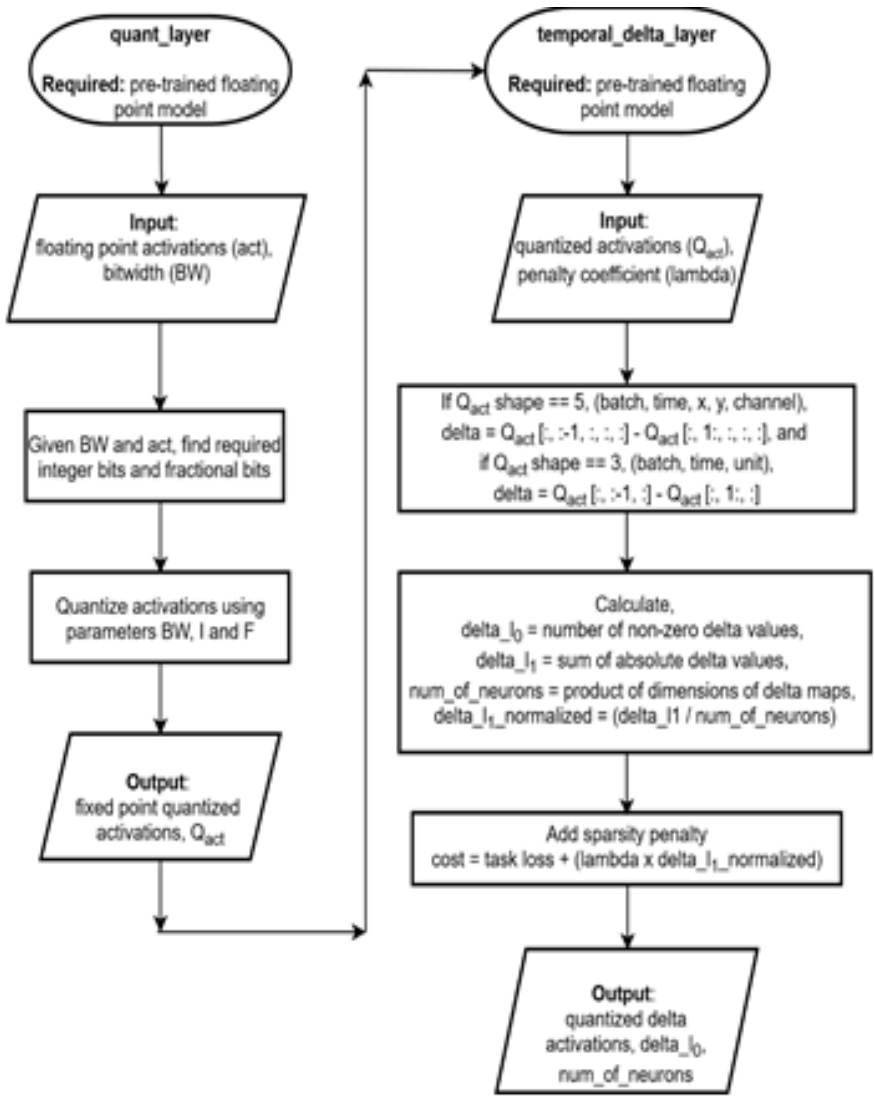

Figure 3.7 Methodology flow of temporal delta layer with fixed point quantization.

The baseline weights were used as a starting point, and all the layers including the temporal delta layer is fine-tuned until acceptable convergence. The hyper-parameters specifically required for this setup were bit width (to which the activations were to be quantized) and penalty co-efficient to balance the tussle between task loss and sparsity penalty

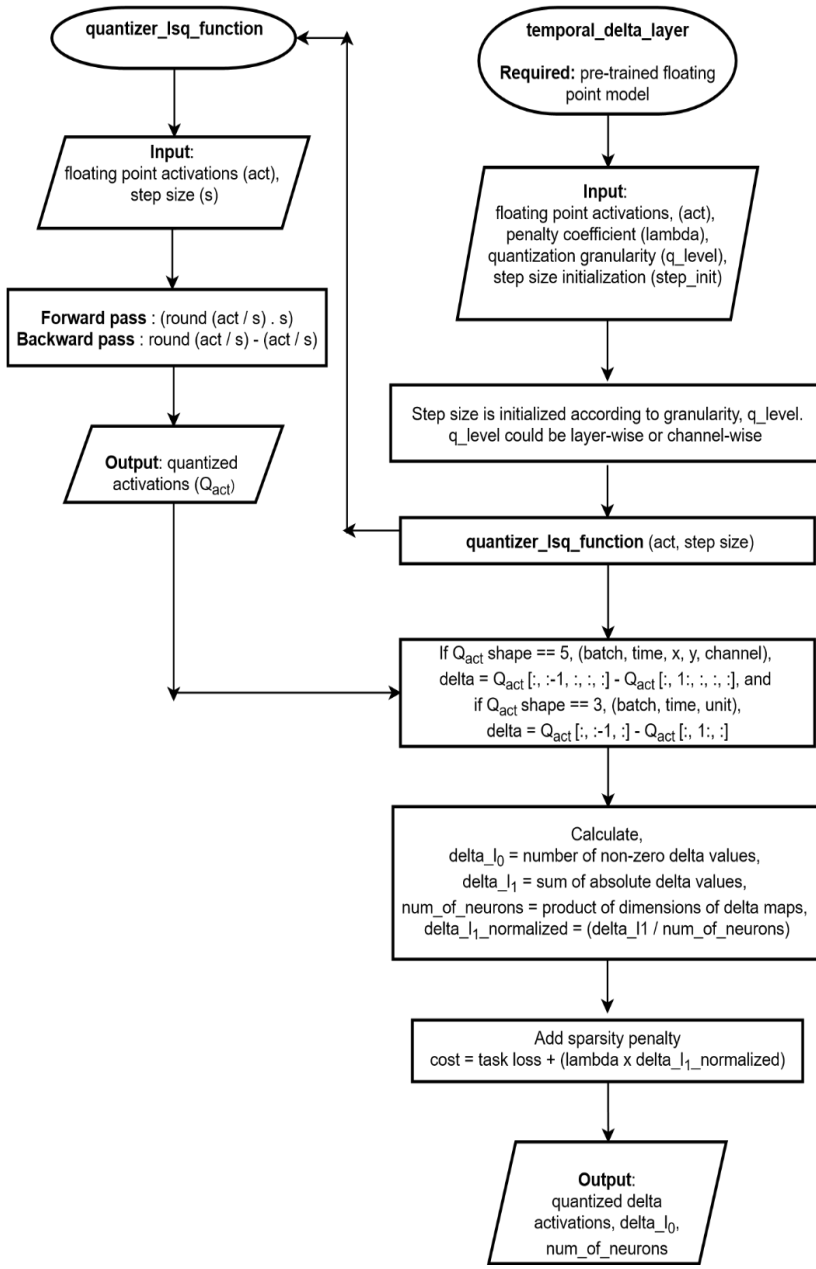

Figure 3.8 Methodology flow of temporal delta layer with learned step size quantization.

Scenario 2: The setup is like the previous scenario except for the activation quantization method used. The previous experiment used fixed precision quantization where all the activation layers in the network were quantized to the same bit width. However, this experiment uses learnable step-size quantization (LSQ), which performs channel-wise quantization depending on the activation distribution resulting in mixed-precision quantization of the activation maps. The layer also introduces a hyperparameter during training (apart from the penalty coefficient mentioned earlier) for the step size initialization. Then, during training, the step size increases or decreases depending on the activation distribution in each channel.

3.3.4.3 Accuracy v/s Activation sparsity

Tables 3.1 and 3.2 show the baseline accuracy and activation sparsity compared against the two scenarios mentioned.

Firstly, when the temporal delta layers with fixed point quantized activations are included in the baseline model, it can be observed that the activation sparsity increases considerably with a slight loss in accuracy. One interesting observation is that, although the sparsity penalty in the temporal delta layer does a good job of decreasing the activation density, quantizing the activations from floating to fixed point representation pushes the activation sparsity of the model even higher. This is because lowering the precision from 32 bits to 8 bits (or less) leads to temporal differences of activations going to absolute zero.

Additionally, the reason for close-to baseline accuracy in the method involving fixed point quantization can be attributed to fractional bit allocation flexibility. That is, as the bit width is fixed, the number of integer bits required is decided depending on the activation distribution within the layer, and the rest of the bits are assigned as fractional bits. This makes sure that the precision of the activation is compromised for range.

Also, another contributing factor for accuracy sustenance is that the first and the last layers of the model are not quantized, similar to works like

Table 3.1 Spatial stream - comparison of accuracy and activation sparsity obtained through the proposed scenarios against the benchmark. In the case of fixed-point quantization, the reported results are for a bit width of 6 bits.

Model setup (Spatial stream)	Accuracy	Activation sparsity
Baseline	75%	48%
Temporal delta layer with fixed point quantization	73%	74%
Temporal delta layer with learned step-size quantization	**69%**	**86%**

Table 3.2 Temporal stream - comparison of accuracy and activation sparsity obtained through the proposed scenarios against the benchmark. In the case of fixed-point quantization, the reported results are for a bit-width of 7 bits.

Model setup (Temporal stream)	Accuracy	Activation sparsity
Baseline	70%	47%
Temporal delta layer with fixed point quantization	68%	67%
Temporal delta layer with learned step-size quantization	**65%**	**89%**

[55–57]. This is because the first and last layer has a lot of information density. Those are the layers where input pixels turn into features and features turn into output probabilities, respectively, which makes them more sensitive to quantization.

Although the activation sparsity gain in the case of the temporal delta layer with fixed point quantization is better than the baseline, it is still not sufficiently high as required. In this effort, the bit-width of the activations are decreased in the expectation of increasing sparsity. However, as the bit-width goes below a certain value (6 bits for spatial and 7 bits for temporal stream), sparsity increases, but accuracy starts to deteriorate beyond recovery, as shown in Table 3.3. This is because quantizing all layers of a network to the same bit-width can mean that the inter-channel variations of the feature maps are not fully accounted for. Since the number of fractional bits is usually selected to cover the maximum activation value in a layer, the fixed bit-width quantization tends to cause excessive information loss in channels with a smaller dynamic range. Therefore, it can be inferred that mixed-precision quantization of activations is a better approach to obtain good sparsity without compromising accuracy.

Table 3.3 Result of decreasing activation bit-width to increase activation sparsity while maintaining accuracy. For spatial stream, decreasing below 6 bits caused the accuracy to drop considerably. For temporal stream, the same happened below 7 bits.

	Spatial stream		Temporal stream	
Activation bit-width	Accuracy (%)	Activation sparsity (%)	Accuracy (%)	Activation sparsity (%)
32	75	50	70	47
8	75	68	70	65
7	75	71	**68**	**70**
6	**73**	**75**	61	73
5	65	80	-	-

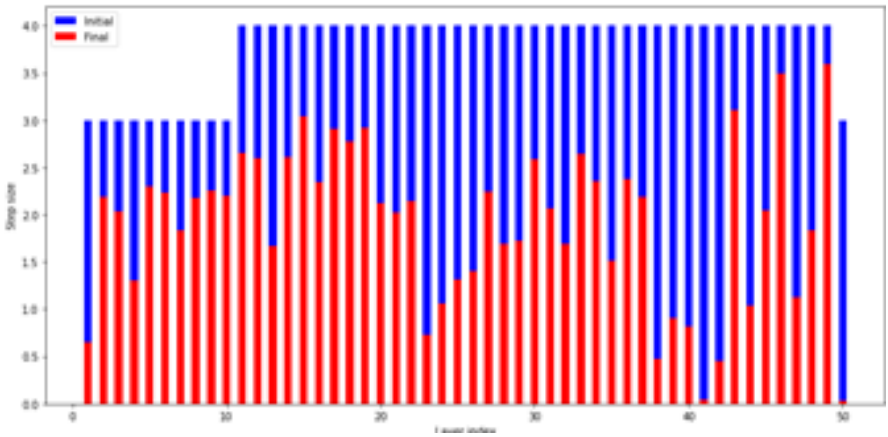

Figure 3.9 Evolution of step size from initialization to convergence. As step-size is a learnable parameter, it gets re-adjusted during training to cause minimum information loss in each layer.

Finally, using the temporal delta layer where incoming activations are quantized using learnable step-size quantization (LSQ) gives the best results for both spatial and temporal streams. As the step size is a learnable parameter, it gives the model enough flexibility to result in a mixed precision model, where each channel in a layer has a bit-width that suits its activation distribution. This kind of channel-wise quantization minimizes the impact of low-precision rounding.

It is also evident in Figure 3.9 that as the training nears convergence, the values of the step size differ according to the activation distribution and bit width required to represent each layer. Moreover, consistent with the literature [58], the first and last layers during training opts for smaller step sizes implying they need more bandwidth for their representation.

The weights generated using this method was then average fused to find the final two-stream network accuracy and activation sparsity (Table 3.3). Finally, the proposed method can achieve 88% activation sparsity with a 5% accuracy loss.

3.4 NN Compiler for Dedicated Inference Accelerator Hardware with Analog In-Memory Computing

This section explains the role of NN compilers in inference hardware accelerator as well as the methodology to follow to implement and evaluate one.

To map the tasks of an NN algorithm to a dedicated hardware with analog in-memory computing for inference, a compiler is needed to automatically generate the instruction set that would provide better performance on the dedicated hardware. The input of such compiler is the trained NN algorithm as well as the hardware architecture of the inference accelerator while the output is the executable set of operations. Therefore, the NN algorithm can only have the type of layers supported on the hardware.

NNs consist of large amounts of Multiplication-and-Accumulation (MAC) operations. Therefore, analog in-memory computation accelerators are ideal to perform such operations. However, different sizes, shapes and bit resolutions make challenging to map NNs on analog crossbar arrays. Nowadays multi-core analog accelerators are becoming very popular for NN inference [64, 65]. Each accelerator core consists of an analog crossbar array surrounded by ADCs, DACs and digital logic (FSMs, etc.). Multiple processing cores are connected via NoC (Network-on-Chip), which is used to transfer data between processing cores. Moreover, analog crossbar arrays are not fixed anymore. State-of-the-art crossbar arrays consist of small crossbar elements which can be horizontally or vertically concatenated using programmable switches. The programmability of crossbar makes them very flexible. The crossbar performs multiplications, and the digital logic configures the crossbar switches and controls data flow (weights and activations, etc.) to the crossbar. However, the flexible architecture and constraints of modern analog accelerators pose a challenge in terms of NN workload, mapping, and scheduling. The traditional mapping techniques such as loop tiling, and loop interchange are not efficient anymore [66]. Moreover, the digital FSMs controlling the crossbar and NoC also require a complex instruction set. To the best of our knowledge, there are no commercial compilers available which can be used to map NN workloads and generate the instruction set for flexible and multi-core analog in-memory accelerators.

3.4.1 Compiler Components

The compiler consists of three main components: hardware architecture, parser, and mapper, see Figure 3.10. The unique architecture and constrains of the dedicated inference accelerators require the compiler to consider hardware specifications and constraints while mapping NN workloads. Therefore, the compiler generates a hardware representation of the accelerator using the specifications. The compiler also contains a parser that parses the information of each NN node and converts it into a specific data structure. By using the

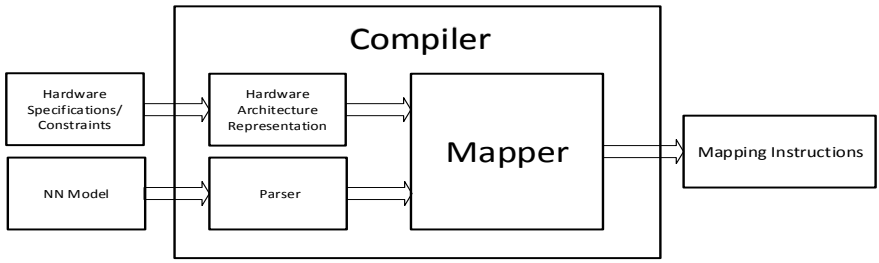

Figure 3.10 Overview of Compiler Tool.

hardware representation and the parsed NN the mapper generates a mapping in the form of instructions for the FSMs.

3.4.2 ONNX Parser

The trained networks are stored using ONNX with a custom export to store additional information such as quantization. The custom ONNX file is parsed by a custom ONNX Parser to extract the information needed by the mapper. This parser parses every graph node of the ONNX model and creates a Python data structure to ease access to information and attributes. The parsed model is a list of nodes and each one stores relevant information. The list of nodes includes:

- Input contains information about the input layer
- Conv contains information regarding convolution layers
- MatMul contains the parameters of fully connected layers
- Add is pointwise addition
- Mul is pointwise multiplication
- Div is a pointwise division
- Act contains activation functions
- Squeeze and Unsqueeze to remove or add singleton dimension when needed.

Moreover, the parser allows the user to fuse the information of consecutive nodes if these nodes follow a certain pattern, e.g., Conv layer followed by one or more of Add, Mul, Div or Act layers are fused together, as shown in Figure 3.11. The fused information is stored in the parsed fused model. The layers are combined when their computations can/should be carried out by same processing core in one computing cycle to minimize data exchange.

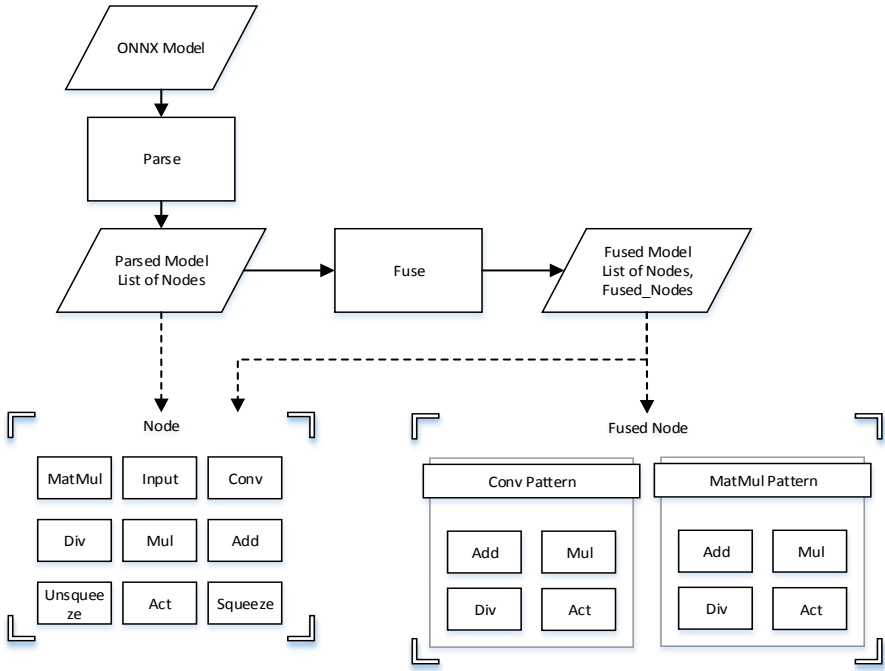

Figure 3.11 ONNX Parser diagram of parsing and fusing the input ONNX model into a list of Nodes and Fused Nodes.

The parsed fused model is used by the compiler to pre-process and generate instructions to run the NN on the hardware.

3.4.3 Hardware Architecture Representation

The hardware architecture representation component allows the compiler to support arbitrary number of processing cores and crossbar array configurations. It generates blueprint of available computation resources for each core according to the configurations and constraints.

The input parameters include the hardware specifications and constrains like crossbar specifications, number of processing cores, memory sizes, etc. When the architectural parser is invoked, it generates a blueprint of the FSMs, the processing cores and the NoC controller. The NoC controller is responsible for data transfer between processing cores and the FSMs for the storage of the weights on the crossbar array, providing the input data to the crossbar array according to the input specifications of each layer and handling the output results of the crossbar array.

3.4.4 Mapper

The mapper analyses the parsed NN and available computation resources on the hardware to map the NN workload into the processing cores. Figure 3.12 shows the flow of the mapper.

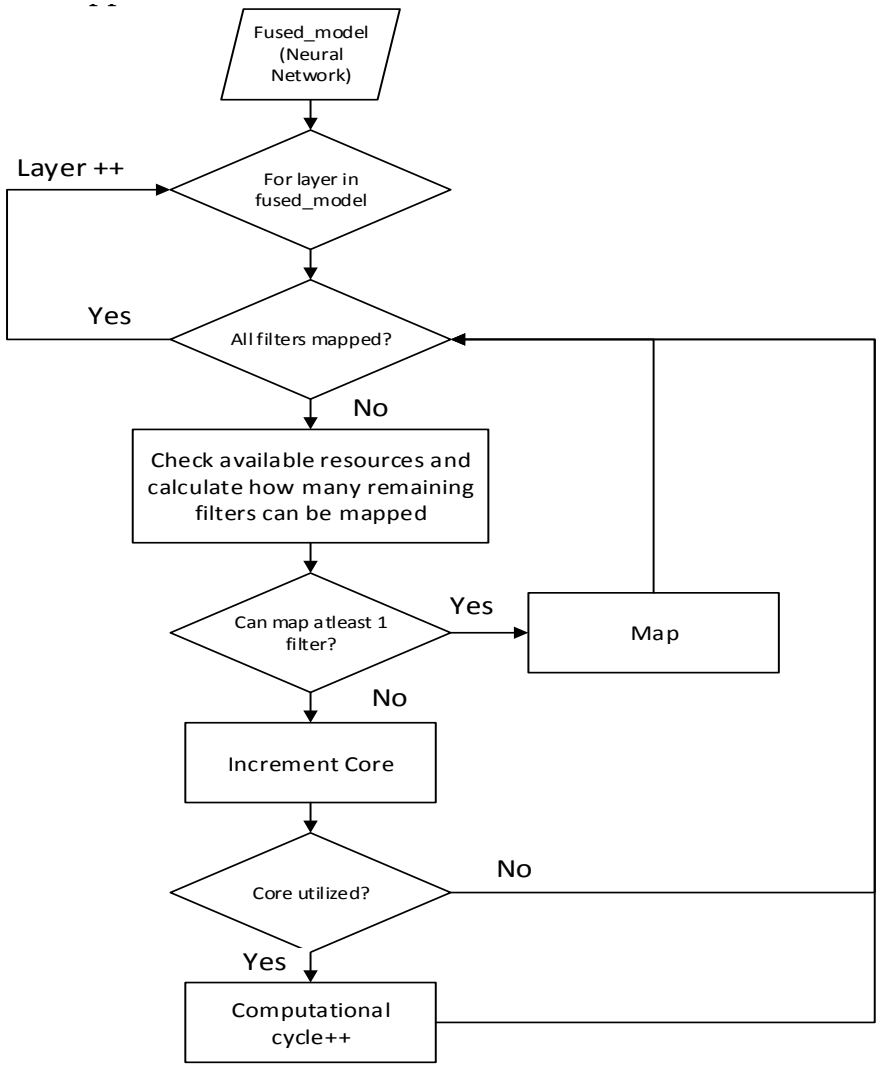

Figure 3.12 Mapping flow of the Compiler.

The mapper maps a NN workload layer by layer. Analog crossbars perform computations using vector-matrix multiplications. Therefore, the compiler converts different NN workloads into vector-matrix multiplications. Moreover, the compiler also schedules the data transfers between processing cores. The weights are converted into a matrix representation and stored in crossbar memory columns and inputs are converted into vectors and fed to crossbar rows using DACs. The mapper determines the required hardware resources for every layer and then it checks how many filters of a layer can be mapped on the current processing core. If there is enough space to map all filters of a layer, then all filters are mapped to the current processing core. Otherwise, a layer is partially mapped to a processing core and remaining filters are mapped to the next core. Similarly, when all processing cores are utilized, the mapper starts mapping again from the first processing core for the next computation. The mapper keeps increasing the computation cycle until the whole NN workload is mapped. When mapper maps the NN workload to a part of processing core, that part is marked as utilized and the parameter values in all FSMs are set according to the mapping. After mapping a workload to each processing core, the compiler generates instructions sets that can be decoded on the dedicated hardware to generate sets of parameters, which are then used to configure the processing cores.

3.4.5 Mapping Strategy

The crossbar array of a processing core is composed of multiple rows and columns of analog synaptic weights performing MAC operations. The crossbar allows to map either fully connected layers or convolutional layers with different kernel sizes.

Figure 3.13 shows the mapping strategy of how a small CNN with two convolution layers has been mapped to a crossbar array of a processing core. Each black rectangle represents 16x4 synaptic weights. The input size is 13x64, the kernel sizes of Conv1 and Conv2 are 4x4x1, with 16 filters, and 3x3x16, with 8 filters, respectively, and the output size is 3x29x8. According to the compiler flow, the compiler checks first for available resources in the processing core and then determines how many synaptic weights are required to map a layer. At the beginning, since the whole processing core is available and the Conv1 filters are small, Conv1 layer can be fully mapped. The filters of Conv1 are converted into vectors and mapped to crossbar like a matrix. The Conv1 filters' vectors are small, and each filter requires 4x4x16 synaptic weights. Similarly, filters of Conv2 are also converted to a vector and mapped.

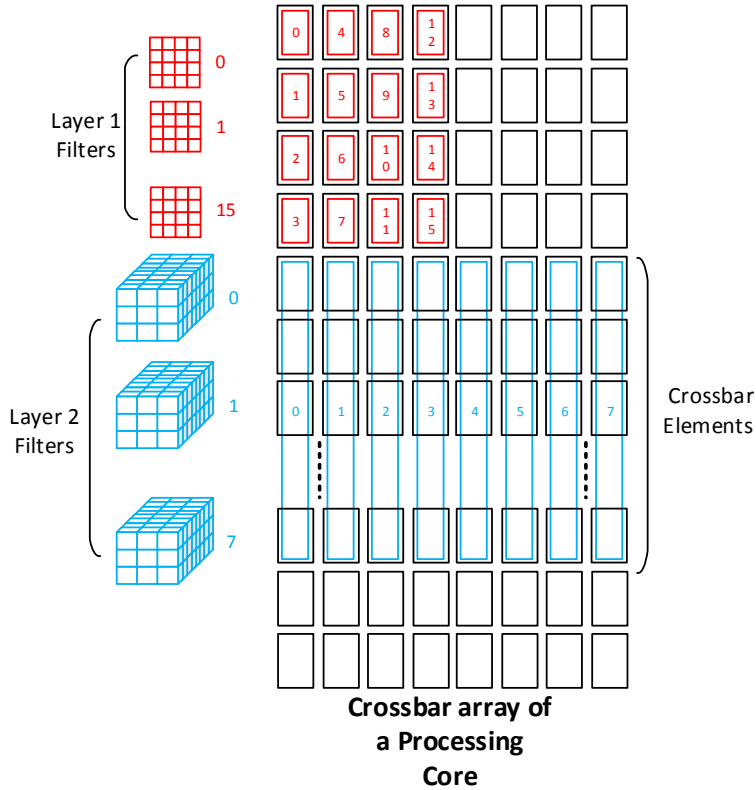

Figure 3.13 Mapping of layers 1 and 2 on processing core 1.

Since Conv2 layer consists of large filters, it requires 3x3x16x8 synaptic weights. The mapper will evaluate if enough synaptic weights are available in the same processing core at which Conv1 was mapped. If it is not the case, a new processing core will be used for mapping Conv2 layer.

3.4.6 Mapping of Deep Spiking NN Architectures to Digital SNN Inference Devices

This section presents an approach for mapping arbitrary deep SNN network architectures onto fixed-architecture inference devices. As example, a device is considered with a single population of hidden neurons, supporting a fixed number of synaptic inputs per neuron, and with a limited number of input and output neurons (see Figure 3.14).

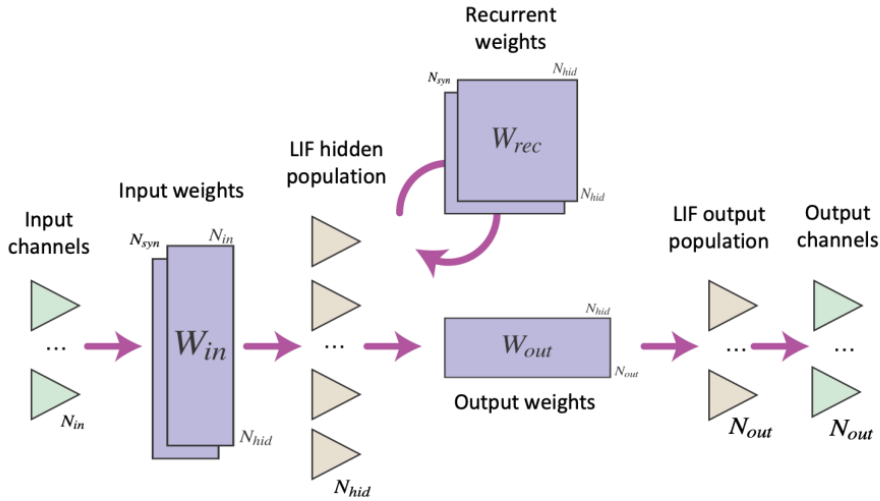

Figure 3.14 A HW architecture for SNN inference.

This architecture supports a fixed maximum number of input channels N_{in}; a fixed maximum number of hidden neurons N_{hid}; and a fixed maximum number of output neurons N_{out}. The neurons are LIF spiking neurons, supporting several synaptic inputs N_{syn}. This architecture implements a single hidden population with the possibility of recurrent weights W_{rec}. Multi-layer networks must be mapped into this single hidden population using the recurrent weights. Other logical weight blocks are supported for input weights W_{in} and output weights W_{out}. These weight matrices are assumed to be sparse, with a limited maximum fan-in N_F per neuron. Only on-zero weights are stored, with weights linearised into memory blocks of fixed maximum size N_F * N. Figure redrawn from [82].

A mapping system must be flexibly to accommodate a wide range of SNN network architectures, including recurrent spiking populations (e.g., reservoir networks and other recurrent architectures; deep feed-forward architectures; and residual network architectures. An example of a deep spiking network making use of all these architectural elements is shown in Figure 3.14. Several LIF spiking neuron layers ("LIF"; orange) are connected via weight blocks ("W"; blue). The first LIF layer "LIF_1" is recurrently connected with weights "W_{rec}". Residual blocks (dashed) include additional connections bypassing the blocks inside.

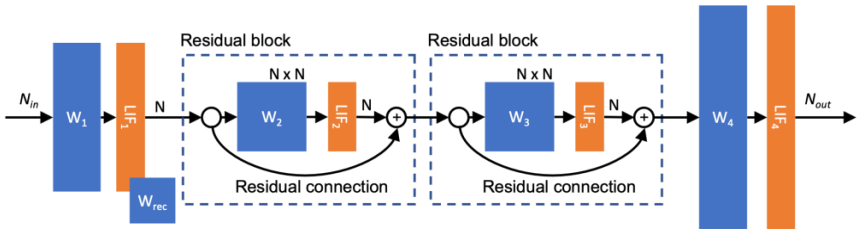

Figure 3.15 An example of a deep spiking network that will be mapped to a HW architecture.

To map the network onto the HW architecture shown in Figure 3.15, several steps are taken.

1. *Graph extraction:* Convert the simulation modules from a high-level representation to a standardised set of graph modules. These graph modules individually represent the weights and neuron populations in the network, as well as embodying a traversable graph that accurately corresponds to the computations and information flow in the network.
2. *DRC check:* Perform a design-rule-check to ensure that the network is compatible with the hardware architecture.
3. *HW mapping:* Assign the network logical resources to available hardware resources.
4. *Parameter configuration:* Assign network parameters to appropriate HW memory blocks and serialise to a bitstream to configure the HW.

3.5 Simulator/Profiler

When implementing a neural network topology on an embedded hardware target, it is critical to do it being able to profile that network and to simulate it, by considering the hardware architecture for extracting the right power/performance/latency figures. Profiling a network means extracting key parameters of interest when fed with representative data. This obviously relates to the number of parameters and number of operations, which is readily available for a given topology, but not only. To choose the right hardware target and optimize the mapping of the network or its graph transformation, the data volume and data bandwidth per layer are also needed. Counter intuitively, the highest data bandwidth does not occur in the layers with the higher number of parameters. This is illustrated in the figures below, considering a popular network topology for embedded applications, i.e., a MobileNet V1.

Figure 3.16 depicts the number of parameters per layer: the deeper the layer, the higher the number of parameters. The depth wise convolution layers use less parameters than the pointwise convolution layers since they use 2D filters instead of 3D ones.

But, as can be seen in Figure 3.17, the first layers are the ones having the biggest data volume. This is because there are more parameters reused on those first layers.

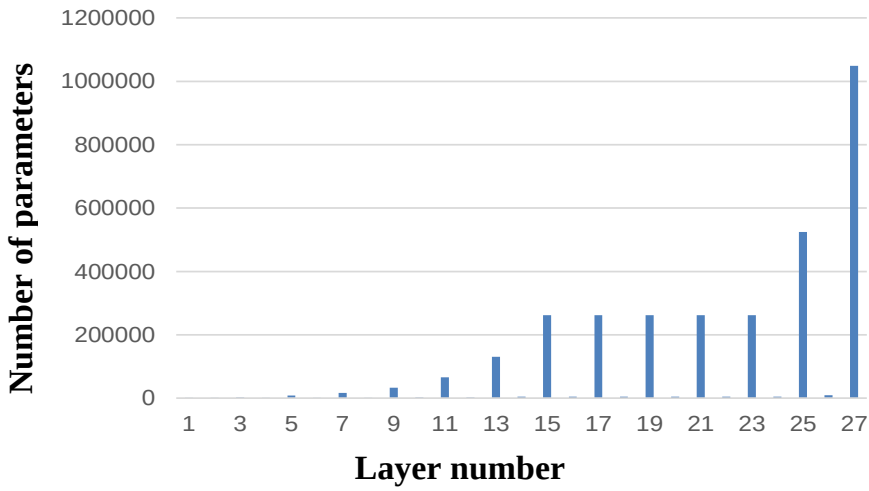

Figure 3.16 MobileNet V1 parameters per layer.

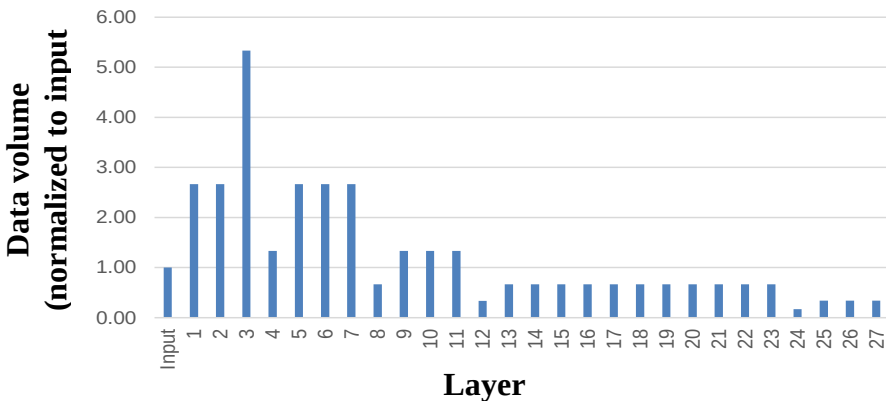

Figure 3.17 MobileNet V1 data volume per layer, normalized to input data volume.

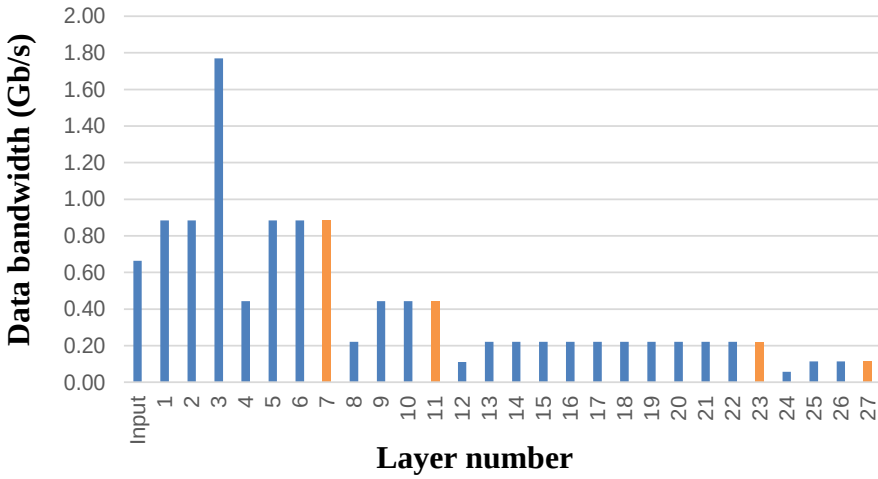

Figure 3.18 MobileNet V1 bandwidth (Gb/s) at each layer.

This profiling shows that it is more important to keep the data locally for layers 1 to 12 to minimize data movement: thus, a data-flow architecture for those layers might be a good fit.

Figure 3.18 also shows the impact of stride on data volume reduction. For instance, layers 1, 4, 8 and 24 have a stride of 2.

The data bandwidth is obviously proportional to the size of the input image and number of images per second. Figure 3.20 illustrates the bandwidth for a 30FPS, 1280x720p, 8b RGB input. The intermediate layers use 4b activations in this case.

Depending on the application, i.e., segmentation, detection, classification, different layers can be exploited. Those layers are highlighted in orange. For object detection, the most important layers are number 27 and 23. For segmentation, layers number 7 and 11 can be exploited, with layer 7 having a higher bandwidth and thus a higher definition.

To obtain those figures, the N2D2 [67, 68] (Neural Network Design & Deployment) dedicated framework is used in ANDANTE for deploying neural networks on digital hardware targets, see Figure 3.19.

For optimizing a network, N2D2 considers applicative performance metrics to be achieved and the hardware target memory capacity. It exploits sparsity of weights by implementing state-of-the-art quantization-aware training methods, such as SAT and LSQ. Finally, N2D2 can address several hardware targets, generating bit streams or configuration files, but it can also be used

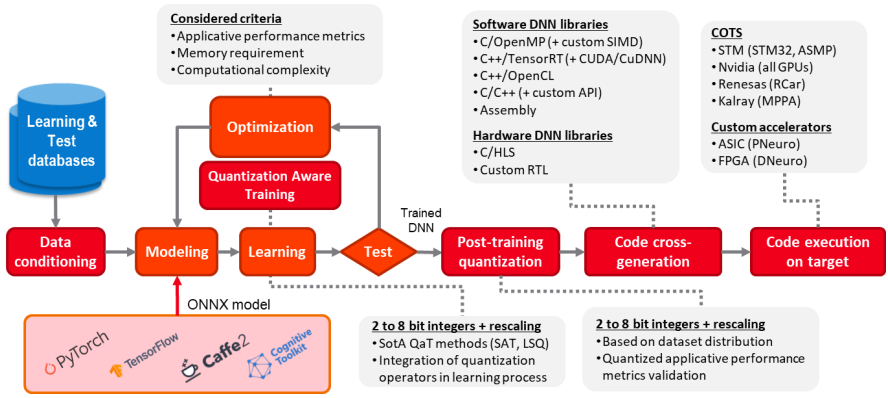

Figure 3.19 N2D2: Neural Network Design & Deployment.

for driving architecture exploration. In such a case, a graph transformation is necessary for accurately performing the simulation and obtaining the Key Parameters of Interest.

Figure 3.20 shows the three steps needed for converting a topology to target a pipelined DNN architecture:

1. The selection of the pre-templated architecture to be used for implementing each layer type (sub-steps a and b in the figure below).

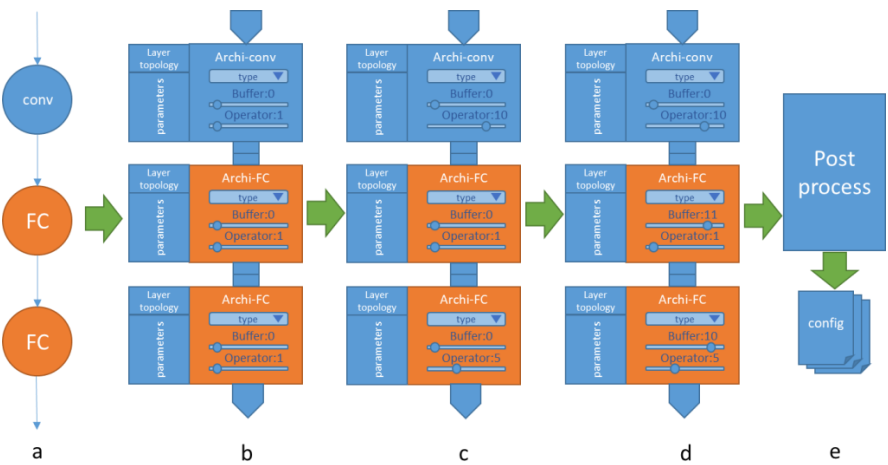

Figure 3.20 Process flow: (a,b) conversion of the neural network to the hardware representation, (c) tuning of the layer parallelism at architectural level, (d) tuning of the buffer, (e) post-processing.

2. The configuration of the resources used, in each architecture, in terms of parallelism and buffer (sub-steps c and d).
3. The network parameters post-process (sub-step e).

This enables to tune various architecture parameters, such as the parallelism of each layer, the buffer, etc. and assess their impact.

The first two steps, architecture selection and hardware resource configuration, are performed using a ROI algorithm. Each layer of the network is associated with a hardware architecture corresponding to the type of operation carried out in the layer. Several architectural models can be considered for a given layer type; in which case the best suited architecture is selected for that layer.

At the end of these steps, a graph of configured architectures is obtained, that ensures the smooth running of the calculation throughout the pipeline. Those "hardware-aware" simulations prove to be much more accurate in terms of energy efficiency and latency, while being bit-equivalent to high level simulations.

3.6 Conclusions

3.6.1 On NN Model Transformation

Intuitively, the new temporal delta layer [63] casts the temporal activation sparsity between two consecutive feature maps into spatial activation sparsity of their delta map. This spatial sparsity is then exploited to reduce computations and memory access when performing sparse tensor multiplications in hardware. As shown in 3.4 proposed method resulted in 88% activation sparsity with an accuracy drop of 5% on UCF-101 dataset for human action recognition.

Table 3.4 Final results on 2 stream networks after average fusing the spatial and temporal stream weights. With 5% accuracy loss, the proposed method almost doubles the activation sparsity available in comparison to the baseline

	Baseline		**Proposed method**	
Model type	Accuracy (%)	Activation sparsity (%)	Accuracy (%)	Activation sparsity (%)
Spatial stream	75	50	69	86
Temporal stream	70	46	65	89
Two-stream (Average fused)	82	47	**77**	**88**

The collateral advantage of temporal sparsity is that the computations does not increase linearly with the increase in frame rate. In standard DNN, doubling the frame rate naturally would require double the computations. However, in the case of temporal delta layer-based model, increasing the frame rate would not only increase the temporal precision of the network but also increase the temporal sparsity limiting the computations required [59].

The drawback of using temporal delta layer derives from its requirement to keep track of the previous activations to perform delta operations. This increases the overall memory footprint which in turn increases the reliance on off-chip memory. For instance, external DRAM memory consumes two orders of magnitude more energy than SRAM [60]. However, the increasing popularity of new memory technologies (like resistive RAM [61], embedded Flash memory [62], etc.) may improve the cost calculations in the near future.

3.6.2 On NN Compiler for Dedicated Inference Accelerator Hardware with Analog In-Memory Computing Conclusion

The main objectives for a compiler tool are maximize hardware utilization, maximize throughput, and minimize latency. However, there is always a trade-off between these objectives since not all of them can be achieved at the same time. Therefore, in the particular methodology described in Section 4 maximum utilization of hardware resources is the focus. The mapping algorithm checks the resources needed for allocating each layer of the NN. Afterwards checks for the available resources on the hardware and tries to find the optimum mapping to fully utilize each processing core.

3.6.3 Simulator/Profiler

Profiling a neural network is essential when considering deploying it on an embedded hardware target. Indeed, for choosing the right target and correctly mapping the network on it, one obviously needs to know the number of parameters (for the memory footprint), the number of operations (for the number of processing elements in a latency-constrained implementation) but also the data bandwidth and data volume. Such a simulation/profiling environment is developed and used in the ANDANTE project. It allows to assess the impact of quantization (even mixed quantization depending on the layers), to identify the most adequate layers for e.g., object detection or image segmentation (in terms of data bandwidth). It can also guide architecture exploration, with a mix of spatially expanded and spatially folded architecture, and the needed graph transformations.

Acknowledgements

This work is supported through the project ANDANTE. ANDANTE has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No. 876925. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Belgium, Germany, Netherlands, Portugal, Spain, Switzerland. ANDANTE has also received funding from the German Federal Ministry of Education and Research (BMBF) under Grant No. 16MEE0116 and 16MEE0111K. The authors are responsible for the content of this publication.

References

[1] ANDANTE. AI for New Devices and Technologies at the Edge. Available online at: https://www.andante-ai.eu/

[2] Capra, M., et al. "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead." *IEEE Access* 8 (2020): 225134-225180.

[3] T. Liang, J. Glossner, L.Wang, and S. Shi, "Pruning and quantization for deep neural network acceleration: A survey," arXiv preprint arXiv:2101.09671, 2021.

[4] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," arXiv preprint arXiv:2102.00554, 2021.

[5] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," International Journal of Computer Vision, vol. 129, no. 6, pp. 1789–1819, 2021.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J Dally, "Eie: Efficient inference engine on compressed deep neural network," ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 243–254, 2016.

[7] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," ACM SIGARCH Computer Architecture News, vol. 45, no. 2, pp. 27–40, 2017.

[8] D. Kim, J. Ahn, and S. Yoo, "Zena: Zero-aware neural network accelerator," IEEE Design & Test, vol. 35, no. 1, pp. 39–46, 2017.

[9] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," IEEE Signal Processing Magazine, vol. 36, no. 6, pp. 51–63, 2019.

[10] M. Mirsadeghi, M. Shalchian, S. R. Kheradpisheh, and T. Masquelier, "Stidi-bp: Spike time displacement based error backpropagation in multilayer spiking neural networks," Neurocomputing, vol. 427, pp. 131–140, 2021.

[11] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event driven networks for image classification," Frontiers in neuroscience, vol. 11, pp. 682, 2017.

[12] M. Sorbaro, Q. Liu, M. Bortone, and S. Sheik, "Optimizing the energy consumption of spiking neural networks for neuromorphic applications," Frontiers in neuroscience, vol. 14, pp. 662, 2020.

[13] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l 0 regularization," arXiv preprint arXiv:1712.01312, 2017.

[14] Z. Liu, J. Li, Z. Shen, G. Huang, S.g Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in Proceedings of the IEEE international conference on computer vision, 2017, pp. 2736–2744.

[15] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in International Conference on Machine Learning. PMLR, 2020, pp. 5533–5543.

[16] G. Georgiadis, "Accelerating convolutional neural networks via activation map compression," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 7085–7095.

[17] A. Yousefzadeh and M. Sifalakis, "Training for temporal sparsity in deep neural networks, application in video processing," arXiv preprint arXiv:2107.07305, 2021.

[18] A. Yousefzadeh, M. A. Khoei, S.r Hosseini, P. Holanda, S. Leroux, O. Moreira, J. Tapson, B. Dhoedt, P. Simoens, T. Serrano-Gotarredona, and B. Linares-Barranco, "Asynchronous spiking neurons, the natural key to exploit temporal sparsity," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 4, pp. 668–678, 2019.

[19] J. Zhao, J. Yang, J. Wang, and W. Wu, “Spiking neural network regularization with fixed and adaptive drop-keep probabilities,” IEEE Transactions on Neural Networks and Learning Systems, 2021.

[20] T. Pellegrini, R. Zimmer, and T. Masquelier, “Low-activity supervised convolutional spiking neural networks applied to speech commands recognition,” in 2021 IEEE Spoken Language Technology Workshop (SLT). IEEE, 2021, pp. 97–103.

[21] D. Neil, M. Pfeiffer, and S.-C. Liu, “Learning to be efficient: Algorithms for training low latency, low-compute deep spiking neural networks,” in Proceedings of the 31^{st} annual ACM symposium on applied computing, 2016, pp. 293–298.

[22] M. Jerry et al., "Ferroelectric FET analog synapse for acceleration of deep neural network training," 2017 IEEE International Electron Devices Meeting (IEDM), 2017, pp. 6.2.1-6.2.4, doi: 10.1109/IEDM.2017.8268338

[23] S., Dutta, C., Schafer, J., Gomez, K., Ni, S., Joshi, and S. Datta, (2020). Supervised learning in all FeFET-based spiking neural network: Opportunities and challenges. Frontiers in Neuroscience, 14, 634.

[24] S., Dutta, V., Kumar, A. Shukla, et al. “Leaky Integrate and Fire Neuron by Charge-Discharge Dynamics in Floating-Body MOSFET”. Sci Rep 7, 8257 (2017).

[25] B. Trevor, et al. Nengo: a Python tool for building large-scale functional brain models, Frontiers in Neuroinformatics , Volume 7, 2014.

[26] M. Arsalan, A. Santra and V. Issakov, "RadarSNN: A Resource Efficient Gesture Sensing System Based on mm-Wave Radar," in IEEE Transactions on Microwave Theory and Techniques, doi: 10.1109/TMTT.2022.3148403.

[27] V. Senft, T. C. Stewart, T. Bekolay, C. Eliasmith, B. J. Kröger, “Reduction of dopamine in basal ganglia and its effects on syllable sequencing in speech”: A computer simulation study, Basal Ganglia, Volume 6, Issue 1, 2016.

[28] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. Van Gool, “Temporal segment networks: Towards good practices for deep action recognition,” in European conference on computer vision, pp. 20–36, Springer, 2016.

[29] K. Chen and W. Tao, “Once for all: a two-flow convolutional neural network for visual tracking,” IEEE Transactions on Circuits and Systems for Video Technology, vol. 28, no. 12, pp. 3377– 3386, 2017.

[30] K. Kang, H. Li, J. Yan, X. Zeng, B. Yang, T. Xiao, C. Zhang, Z. Wang, R. Wang, X. Wang, et al., "T-cnn: Tubelets with convolutional neural networks for object detection from videos," IEEE Transactions on Circuits and Systems for Video Technology, vol. 28, no. 10, pp. 2896–2907, 2017.

[31] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149, 2015.

[32] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," arXiv preprint arXiv:1902.09574, 2019.

[33] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," arXiv preprint arXiv:1503.02531, 2015.

[34] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," arXiv preprint arXiv:1608.03665, 2016.

[35] M. Mahowald, "The silicon retina," in An Analog VLSI System for Stereoscopic Vision, pp. 4– 65, Springer, 1994.

[36] J. W. Mink, R. J. Blumenschine, and D. B. Adams, "Ratio of central nervous system to body metabolism in vertebrates: its constancy and functional basis," American Journal of PhysiologyRegulatory, Integrative and Comparative Physiology, vol. 241, no. 3, pp. R203–R212, 1981.

[37] A. Yousefzadeh, M. A. Khoei, S. Hosseini, P. Holanda, S. Leroux, O. Moreira, J. Tapson, B. Dhoedt, P. Simoens, T. Serrano-Gotarredona, et al., "Asynchronous spiking neurons, the natural key to exploit temporal sparsity," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 4, pp. 668–678, 2019.

[38] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 21–30, 2018.

[39] O. Moreira, A. Yousefzadeh, F. Chersi, G. Cinserin, R.-J. Zwartenkot, A. Kapoor, P. Qiao, P. Kievits, M. Khoei, L. Rouillard, et al., "Neuronflow: a neuromorphic processor architecture for live ai applications," in 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 840–845, IEEE, 2020.

[40] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," arXiv preprint arXiv:1803.03635, 2018.

[41] H. Yang, W. Wen, and H. Li, "Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures," arXiv preprint arXiv:1908.09979, 2019.

[42] S. Seto, M. T. Wells, and W. Zhang, "Halo: Learning to prune neural networks with shrinkage," in Proceedings of the 2021 SIAM International Conference on Data Mining (SDM), pp. 558–566, SIAM, 2021.

[43] G. Georgiadis, "Accelerating convolutional neural networks via activation map compression," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 7085–7095, 2019.

[44] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, B. Nell, N. Shavit, et al., "Inducing and exploiting activation sparsity for fast neural network inference," in 37th International Conference on Machine Learning, ICML 2020, vol. 119, 2020.

[45] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: A dť ej ‘a vu-free differential deep neural network accelerator," in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 134–147, IEEE, 2018.

[46] C.-Y. Wu, M. Zaheer, H. Hu, R. Manmatha, A. J. Smola, and P. Krĺahenbuĺhl, "Compressed video action recognition," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 6026–6035, 2018.

[47] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, "Eva2: Exploiting temporal redundancy in live computer vision," in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 533–546, IEEE, 2018.

[48] L. Cavigelli, P. Degen, and L. Benini, "Cbinfer: Change-based inference for convolutional neural networks on video data," in Proceedings of the 11th International Conference on Distributed Smart Cameras, pp. 1–8, 2017.

[49] P. O'Connor and M. Welling, "Sigma delta quantized networks," arXiv preprint arXiv:1611.02024, 2016.

[50] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," Sensors, vol. 21, no. 9, p. 2984, 2021.

[51] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," arXiv preprint arXiv:1902.08153, 2019.

[52] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," arXiv preprint arXiv:1806.08342, 2018.

[53] Y. Yu, R. Hira, J. N. Stirman, W. Yu, I. T. Smith, and S. L. Smith, "Mice use robust and common strategies to discriminate natural scenes," Scientific reports, vol. 8, no. 1, pp. 1–13, 2018.

[54] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," arXiv preprint arXiv:1406.2199, 2014.

[55] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," arXiv preprint arXiv:1212.0402, 2012.

[56] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," arXiv preprint arXiv:1805.06085, 2018.

[57] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," arXiv preprint arXiv:1606.06160, 2016.

[58] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," The Journal of Machine Learning Research, vol. 18, no. 1, pp. 6869–6898, 2017.

[59] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in European conference on computer vision, pp. 525–542, Springer, 2016.

[60] M. A. Khoei, A. Yousefzadeh, A. Pourtaherian, O. Moreira, and J. Tapson, "Sparnet: Sparse asynchronous neural network execution for energy efficient inference," in 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), pp. 256–260, IEEE, 2020.

[61] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," Proceedings of the IEEE, vol. 105, no. 12, pp. 2295–2329, 2017.

[62] S. Huang, A. Ankit, P. Silveira, R. Antunes, S. R. Chalamalasetti, I. El Hajj, D. E. Kim, G. Aguiar, P. Bruel, S. Serebryakov, et al., "Mixed precision quantization for reram-based dnn inference accelerators," in 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 372–377, IEEE, 2021.

[63] M. Kang, H. Kim, H. Shin, J. Sim, K. Kim, and L.-S. Kim, "S-flash: A nand flash-based deep neural network accelerator exploiting bit-level sparsity," IEEE Transactions on Computers, 2021.

[64] P. Vijayan, “Temporal Delta Layer.” Available online at: http://resolver.tudelft.nl/uuid:0806241d-9037-4094-a197-6e65d6482f2b.

[65] C. Yakopcic, T. M. Taha and R. Hasan, "Hybrid crossbar architecture for a memristor based memory," NAECON 2014 - IEEE National Aerospace and Electronics Conference, 2014, pp. 237-242. doi: 10.1109/NAECON.2014.7045809.

[66] X. Wang et al., "TAICHI: A Tiled Architecture for In-Memory Computing and Heterogeneous Integration," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 69, no. 2, pp. 559-563, Feb. 2022. doi: 10.1109/TCSII.2021.3097035.

[67] A. Parashar et al., "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019, pp. 304-315. doi: 10.1109/ISPASS.2019.00042.

[68] C. Gamrat “Neural Networks Design and Deployment N2D2 for embedded AI” www.youtube

[69] CEA-LIST/N2D2 -GitHub, github.com > CEA-LIST > N2D2

[70] H., Cai, C., Gan, T., Wang, Z., Zhang, and S. Han, (2019). Once-for-all: Train one network and specialize it for efficient deployment. arXiv preprint arXiv:1908.09791

[71] J. A., Pérez-Carrasco, B., Zhao, C., Serrano, B., Acha, T., Serrano-Gotarredona, S., Chen, and B. Linares-Barranco, (2013). Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward ConvNets. IEEE transactions on pattern analysis and machine intelligence, 35(**??**), 2706-2719.

[72] P. U., Diehl, D., Neil, J., Binas, M., Cook, S. C., Liu, and M. Pfeiffer, (2015, July). Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In 2015 International joint conference on neural networks (IJCNN), pp. 1-8.

[73] D., Zambrano, and S. M. Bohte, (2016). Fast and efficient asynchronous neural computation with adapting spiking neural networks. arXiv preprint arXiv:1609.02053.

[74] S., Kim, S., Park, B., Na, and S. Yoon, (2020, April). Spiking-yolo: spiking neural network for energy-efficient object detection. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 07, pp. 11270-11277).

[75] B., Han, G., Srinivasan, and K. Roy, (2020). RMP-SNN: Residual membrane potential neuron for enabling deeper high-accuracy and

low-latency spiking neural network. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 13558-13567).

[76] S., Deng, and S. Gu, (2021). Optimal conversion of conventional artificial neural networks to spiking neural networks. arXiv preprint arXiv:2103.00476.

[77] S., Narduzzi, S. A., Bigdeli, S. C., Liu, and A. L. Dunbar, (2022). Optimizing the consumption of spiking neural networks with activity regularization. arXiv preprint arXiv:2204.00607

[78] A., Howard, M., Sandler, G., Chu, L. C., Chen, B., Chen, M., Tan, and H. Adam, (2019). Searching for mobilenetv3. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 1314-1324)

[79] E., Hebert Robbins. ASA, A Stochastic Approximation Method, Annals of Mathematical Statistics, 1951, volume 22, pp. (400-407)

[80] J. Kiefer and J. Wolfowitz}, Stochastic Estimation of the Maximum of a Regression Function, Annals of Mathematical Statistics, 1952, volume 23, pp. (462-466), https://doi.org/10.1214/aoms/11[729392

[81] F. Rosenblatt, (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

[82] Kingma and Ba 2014. https://doi.org/10.48550/arXiv.1412.6980

[83] D., Muir, F., Bauer, and P., Weidel (2019). Rockpool Documentaton. Zenodo. 10.5281/zenodo.3773845

[84] B. J. Kröger et al., "Modeling speech production using the neural engineering framework," in Proc. 5th CogInfoCom, Nov. 2014, pp. 203–208

[85] J. A. K. Ranjan et al., "A novel and efficient classifier using spiking neural network," J. Supercomput., vol. 76, pp. 6545–6560, May 2019