POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Embedded systems

Master's Degree Thesis

An Efficient Hardware Accelerator for Class Incremental Deep Neural Networks

Supervisors

Prof. Guido MASERA

Dott. Alberto MARCHISIO

Candidate

Eugenio RESSA

December 2023

Abstract

Machine learning (ML) and neural networks (NN) have great potential in many fields, including the classification of images, sounds, signals, etc. Some recognition and classification tasks, such as speech and image recognition, would not be possible with classical algorithms, but only through ML algorithms. However, ML algorithms are very heavy from a computational, memory and energy consumption point of view, which therefore makes them unsuitable for embedded systems, which are equipped with processors in the order of MHz and a few MB of memory. . For this reason, many HW accelerators have been developed, which manage to maintain low power consumption and reduce computation times by hundreds of times. Nonetheless, these accelerators are static in the tasks they can perform, in the sense that they cannot be trained further to learn and therefore increase the number of executable tasks, without forgetting the previously learned tasks. This is where studies on Continual Learning (CL) arise, i.e. the ability to increase the quantity of accelerator tasks that can be performed, such as, for example, the number of images that can be recognized and classified in image recognition. Studies on CL have proposed various algorithms that allow the NN to continue training with new tasks, preventing it from forgetting previously learned tasks. One of these, GDumb, managed to obtain among the best results on datasets such as CIFAR 10,100 and MNIST, saving the training images in a homogeneous number: in the memory designed to save a part of the training data, each class must have the same number of images, so that after a training cycle, the network will learn the various tasks equally. The objective of this thesis is therefore to develop a general accelerator for image recognition capable of increasing the number of recognizable classes, using the GDumb algorithm. Image classification occurs using weighted convolutional algorithms, whose weights are optimized and trained to achieve the best possible accuracy. In the case of a class increase request to be acknowledged, an optimization loop will be executed to re-train the weights. The accelerator integrates a series of multiply and accumulate, memories designed to store weights and images for training, a control unit that coordinates the inference and training phases. The implementation of the proposed accelerator, synthesized using a 65nm CMOS technology node using the ASIC design flow, achieves 6x reduction of training and inference time, compared to the original software-level algorithm running on a NVIDIA Tesla P100 GPU

Acknowledgements

Giungendo alla fine di questo viaggio, di questa trasformazione, non posso che essere contento di come sia stato.
Felice perchè pieno di bei ricordi, di sfide e di momenti indimenticabili.
Ognuno di questi ricordi è associato a persone a me care, che voglio qui ringraziare, per esser stati compagni di viaggio.
Ai miei genitori e nonni, fonte inesauribile di forza e sostegno, amore e consiglio, sempre presenti, e per questo, insostituibili.
Alla mia metà, che ha diradato le nubi e indicato la via, portando gioia anche nei momenti più duri.
Agli amici, vecchi e nuovi che siano, che han portato risate e spensieratezza, auguro che il tempo non rompa alcun legame.
E per concludere, un ringraziamento va a chi mi ha seguito in questi ultim mesi, portando a conclusione questo percorso.

A tutti voi,
Grazie.

Summary

Introduction

In recent years, Deep Neural Networks (DNNs) have been deployed in several applications, like computer vision, finance, healthcare, and robotics. A common practice is to train a DNN on the desired task using the training set, and then deploy the trained model on the target for inference. In this way, it is possible to conduct DNN training on large data centers (e.g., using high-end GPUs), and then conduct resource-constrained optimizations (e.g., compression) to deploy DNN inference on autonomous systems. However, this practice limits the dynamic capabilities of DNNs that cannot adapt to new tasks or a distribution variation of the input data within the same task. In this regard, the Continual Learning (CL) paradigm enables the dynamic change of DNN parameters to evolve and learn new tasks (or new classes). The main goal of CL-based algorithms is to avoid Catastrophic Forgetting (CF), i.e., the DNN should maintain the knowledge of how to perform the previous tasks while learning new tasks. Since they require the execution of the backpropagation and parameters update, CL algorithms impose more demanding computation and memory resources than traditional systems that conduct only inference. Hence, it is extremely important to execute CL algorithms in an efficient manner to be able to deploy them onto resource-constrained autonomous systems. Most of the existing architectures and optimizations for DNNs focus only on optimizing the inference process. Therefore, they cannot execute the backward operations required by CL algorithms. Some architectures that have been proposed to accelerate the training can be adopted to conduct backpropagation computations, but they do not support the execution of CF-avoiding policies.

Proposed Solution

To overcome these limitations, I propose TinyCL, a hardware architecture that can efficiently execute CL operations on autonomous systems. The system is composed of a Convolutional Neural Network (CNN) able to execute image classification, and a memory, used to retain old training samples to be used for further training, in case new tasks have to be learned.

Figure 1 shows an overview of the system, where a Processing Unit (PU) is connected to 4 different memories. A Training Data Memory containing a subset of the original training memory and implementing the GDumb CL method. The GDumb expects that in the memory the number of training samples is homogeneous among all classes. The Kernel Data Memory contains the parameters of the CNN and the training process will tune them. In the Gradient Memory, the Gradient Propagation is saved during the Stochastic Gradient Descendent (SGD) algorithm

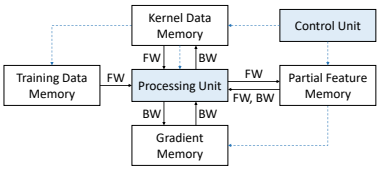

Figure 1: Overview of the TinyCL architecture

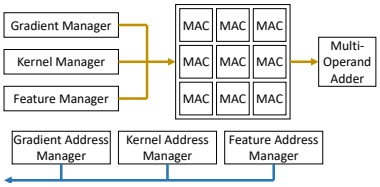

Figure 2: Overview of Processing Unit architecture

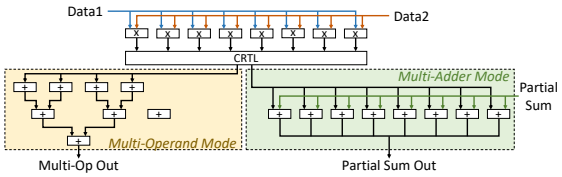

Figure 3: Overview of MAC architecture.

execution used to train my CNN. The Partial Feature Memory is used to temporarily save the output of each layer operation, in order to be used again during SGD operation. The CU manages the data transfer between the PU and the memories. It is also in charge of dynamically adapting the PU to accept different feature sizes, necessary to execute more convolution in series. The PU is composed of nine Multiply and Accumulate, a multi-operand adder, the managers that prepare the Gradient, the Kernel and the Feature to be forwarded to the MACs, and the address managers to address the Gradient, the Kernel, and the Partial Feature memories. Each MAC is composed of 8, 16-bit fixed-point multipliers and 8, 16-bit fixed-point adders. The adders's connection is dynamic, depending on which operation is executed. During Forward and Gradient Propagation computation, the results of the multiplications need to be summed together in order to execute a 3D convolution. Instead, during Kernel Gradient computation, each multiplication refers to a different kernel gradient executed in parallel, and so the 8 adders are connected in parallel, in order to execute 8 different additions. During Forward operation, the multi-channel sliding window is convoluted with a 3D kernel: 72 pixels are multiplied with 72 kernels and summed together each clock cycle, in order to compute one pixel of the output feature. If the input feature has more than 8 channels, the operation is repeated. Due to data reuse (6 pixels out of 9 are shared between two consecutive sliding windows) at each c.c. only 3 pixels are read. The Gradient propagation is computed by executing a convolution with the input gradient propagated from the previous layer and the kernel. The MACs usage is in Multi-Operand mode and 3 pixels are read at each c.c. To compute the Gradient of the Kernel, each MAC computes and accumulates the product between the Input Feature and the propagation of the Gradient. Each MAC computes in

parallel the gradient of 1 pixel of the kernel for 8 channel. A the end of the process, the Kernels are updated with a scaled value of the Kernel Gradient (scaled by the learning rate).

Results

The TinyCL HW accelerator where designed in SystemVerilog and synthesized using a 65 nm library. To validate the behavior of the SV, a Python code where developed to test the Convolutional operations. The Python code was tested implementing 2 CNN: a Double Layer (DL) Convolutional layer + ReLU + Convolutional Layer + ReLU + Dense Layer + Softmax and a Single Layer (SL) Convolutional Layer + ReLU + Dense Layer + Softmax. In both models, the data format was 16-bit fixed point, as in the HW. The accuracy of the software was compared with the same model developed using the TensorFlow library and the results are available in Table 1.

Architecture	dataset	training epoch	Accuracy
SL (mine)	CIFAR10	10	**38%**
SL (TF)	CIFAR10	10	40%
DL (mine)	CIFAR10	10	**49%**
DL (TF)	CIFAR10	10	54%

Table 1: Comparision between my Python code implementing a convolutional NN from scratch and the same NN of TF.

The TinyCL has a die size of 4.72 mm^2, a power consumption of 86 mW, and a critical path time T_{cp} of 3.87 ns. The HW acceleration is compared to the equivalent software-level implementation on TensorFlow of the DL model, running on an NVIDIA Tesla P100. In a CL scenario with 1000 samples Training Data memory, a 10 epoch training was performed in 40 s by the GPU, while TinyCL took 0.86 s, with a speed-up of 46x.

Conclusions

In this work, I show how we can speed up CNNs by accelerating and optimizing the Convolution layer operation in a CL scenario. I show how we can reduce Data memory access by exploiting data reuse. This can reduce data access from 9 to 3 c.c. each. The prefetching and the memories port-wide let the system run without stalling, despite more than one data location is needed. This architecture leads to a 46x speed-up of CNNs, thanks to convolutional layer acceleration, compared to NVIDIA Tesla P100.

Table of Contents

Chapter 1

Introduction

Machine learning is an evolving technology that in recent years has been deployed in several applications, like finance, healthcare, and computer vision. In Machine Learning (ML) a Deep Neural Network (DNN) is used to elaborate data and to output an inference. The task of Machine Learning is to tune and optimize the parameters of the DNN in order to reduce the inference error as much as possible. This process to reduce the error is called training and one of the most used methods to train a DNN is called Stochastic Gradient Descent (SGD), where the gradient of the loss is propagated backward through the DNN in order to change the weights in order to cause a reduction of the loss. A common practice is to train a DNN using a training set of images till the DNN reaches a good value of inference accuracy on another set of images, called Test Set Images. When the training is completed, it will be used to infer, without further training. In this way, it is possible to conduct DNN training on large data centers (e.g., using high-end GPUs) in order to speed up the training, and then conduct resource-constrained optimizations (e.g., compression) to deploy DNN inference on autonomous systems [1].

However, this practice limits the dynamic capabilities of DNNs that cannot adapt to new tasks or a distribution variation of the input data within the same task. For instance, considering the previous image recognition example, if we want to increase the number of classes identified from 10 to 20, a new training cycle shall be executed to reach an acceptable accuracy, using both the old and the new Training Set Images. In this regard, the Continual Learning (CL) paradigm enables the dynamic change of DNN parameters to evolve and learn new tasks (or new classes) [2]. The main goal of CL-based algorithms is to avoid Catastrophic Forgetting (CF), i.e., the DNN should maintain the knowledge of how to perform the previous tasks while learning new tasks [3].

1.1 Target research problems

Since they require the execution of the backpropagation and parameters's update, CL algorithms impose more demanding computation and memory resources than traditional systems that conduct only inference [4]. Hence, it is extremely important to execute CL algorithms in an efficient manner to be able to deploy them onto resource-constrained autonomous systems. Most of the existing architectures and optimizations for DNNs focus only on optimizing the inference process [5]. Therefore, they cannot execute the backward operations required by CL algorithms. Some architectures [6] that have been proposed to accelerate the training can be adopted to conduct backpropagation computations, but they do not support the execution of CF-avoiding policies.

1.2 Novel contribution

To overcome these limitations, we propose TinyCL, a hardware architecture that can efficiently execute CL operations on autonomous systems. Our architecture reuses the same processing units for computing the forward and backward computations, and a specialized control unit dictates the data flow based on the CL policy. In a nutshell, my contributions are :

- RTL Design of the complete TinyCL architecture, in which multiple Processing Units execute the computation in parallel and the convolutional sliding window is designed following a snake-like pattern.
- Synthesis of the TinyCL architecture using the conventional ASIC flow for a 65 nm CMOS technology node.
- Compared to its equivalent software-level implementation on an Nvidia Tesla P100 GPU, the TinyCL architecture achieves 46× speedup; compared to other DNN training accelerators in the literature, the TinyCL architecture achieves lower latency, power consumption, and area, thus making it suitable for being adopted on resource-constrained autonomous systems.
- The complete RTL of the TinyCL architecture will be open-sourced for reproducible research.

Chapter 2

Background and Related Work

2.1 Deep Neural Network in Image Recognition

We can refer to image recognition as the task of classifying an image with a label. This classification operation called *inference*, can be seen as a function $y = f(x, W)$ where y is the inference to which class the image belongs, x is the input image and W is the set of parameters. As a class, in Image Recognition (IR), we mean what is depicted by the image: a dog, a cat, or a car, these are classical classes of images that we would like to identify and classify.

As previously said, we can represent a DNN as $f(x, W)$, where x is the Input Image, called Feature, that has to be classified, while W is a set of parameters. Such parameters are no more than numbers (generally decimal numbers) that

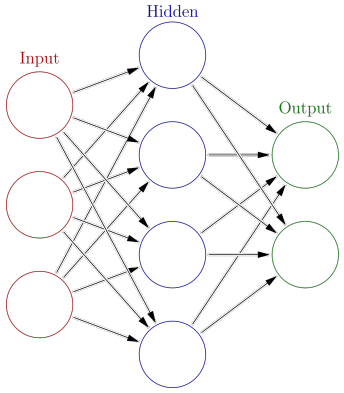

Figure 2.1: Example of a classical multi-layer DNN

are multiplied by the pixels of the image. Figure 2.1 depicts a little DNN called MLP: it is composed of 3 layers where the first is composed of 3 of the so-called "neurons", the second layer of 4 neurons, and the third layer of 2. Each of these neurons can be described as a function $f^{'}(x_i, W_i)$. The function uses several inputs and weights to compute an output y, that is forwarded to the next layer. What exactly these functions are will be seen later. Therefore, by stacking several layers that compute a function we can build a DNN that shall be able to execute an ML task. In image recognition tasks, the DNN infer which image is depicted by the image. As previously said, input data are images represented in RGB or Greyscale notation, generally normalized to values between 0 and 1. The inference process takes place by applying $f(x, W)$ to the feature. As output, the NN gives an array y of cardinality z, where z is the number of classes to be inferred.
y contains the response of the network in the form of an array of probabilities. The higher probabilities is the inference of the DNN of which class the feature is.
However, to make the DNN infer correctly, the parameter of the DNN must be

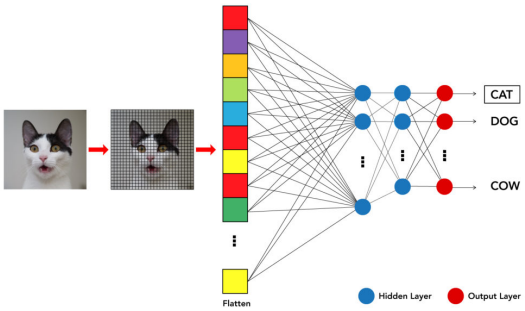

Figure 2.2: Inference in image recognition

tuned in order to make the prediction of the DNN $y^{'}$ as much equal as possible to the real result y. Figure 2.2 shows a classic DNN inference operation: as previously said, the image is used as a matrix and computed by the DNN together with weights. Suppose that this DNN was trained to recognize 10 different animals from the images (cat, dog, cow, etc). Suppose that our DNN has to infer a cat, as shown in Figure 2.2 The inference process of the DNN will output an array of probabilities $y^{'}_{0-9}$, while the correct answer (supposing the probability of the cat is at position 0 of the array) is depicted in 2.1.

$$y_{0-9} = (1,0,0,0,0,0,0,0,0,0) \tag{2.1}$$

The difference between the inference result and the correct result is called loss and we could define a first and easy function to compute it, as in 2.2.

$$L = \sum_{i}^{9} y_i^{'} - y_i \tag{2.2}$$

2.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a branch of DNNs used in image recognition tasks. CNNs are characterized by the presence of a Convolutional Layer, together with other layers like Dense layer, Activation layer, Batch normalization layer and Softmax layer.

2.2.1 Convolutional Layer

In a Convolutional layer, a multi-channel input feature V is convoluted with a 4D kernel K to produce a multi-channel output Z (see 2.3).

$$Z(i,j,k) = c(K,V,s)_{i,j,k} = \sum_{l,m,n} [V(l,(j-1)\cdot s+m,(k-1)\cdot s+n)\cdot K(i,l,m,n)] \quad (2.3)$$

Applying the Stochastic Gradient Descent (SGD), the computation to propagate the gradient across layers is a convolution between the previous kernel K and the gradient G propagated from the previous layer, as shown in 2.4, while the gradient of the kernel is computed through 2.5.

$$h(K,G,s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(V,K) = \sum_{l,m \ s.t.(l-1)\cdot s+m=j} \left(\sum_{n,p \ s.t.(n-1)\cdot s+p=k} \left(\sum_{q} G_{q,l,n} \cdot K_{q,i,m,p} \right) \right) \quad (2.4)$$

$$g(G,V,s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k}} J(V,K) = \sum_{m,n} G_{i,m,n} \cdot V_{j,(m-1)\cdot s+k,(n-1)\cdot s+l} \quad (2.5)$$

2.2.2 Activation Layer

Activation functions are functions used to cut useless information. These functions are applied element-wise after a convolution operation. Taking inspiration from neurons, the activations function decides if a neuron reached a value enough high to be valuable.

Figure 2.3 shows the Relu = max(0,z) activation function, which filters all negative values to zero, maintaining the positive as they are. This behavior is derived from one of the neurons: when the sum of the input signals is strong enough to overcome a barrier, the neuron will "fire", i.e. it got activated.

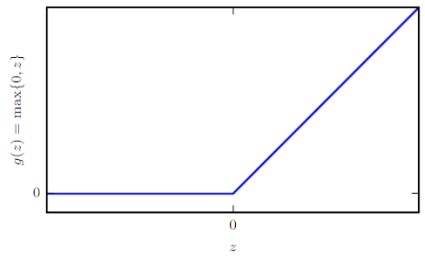

Figure 2.3: ReLU activation layer

2.2.3 Batch Normalization Layer

To increase speed and reduce the difficulty of training a DNN, [7] created the Batch Normalization Layer. Previously we could not use a high learning rate and we must choose carefully the initialization parameters because of internal covariance shift, inserting a layer that normalizes the values across the mini-batch will speed up training because it enables to increase the Learning Rate, without losing the capability to reach a global minima. The equation 2.6 shows the computation of the batch normalization layer, where $E[x^k]$ and $Var[x^k]$ are the mean and the variance of the pixel ^{k} over the mini_batch.

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}} \tag{2.6}$$

2.2.4 Dense Layer

The Dense layer computes a matrix multiplication between a 1D row matrix I with a 2D weight matrix W (see 2.7).

$$y_n = \sum_{i=0}^{m} I_i \cdot W_{i,n} \tag{2.7}$$

Applying SGD, we can compute the gradient propagation that is a matrix multiplication between the input gradient dY propagated from the previous layer and the transpose of the weight (see 2.8).

$$dX_i = \sum_{n=0}^{N} dY_n \cdot W_{n,i}^T \tag{2.8}$$

Then, the gradient of the weights is computed as in 2.9.

$$dW_{i,n} = I_i \cdot dY_n \tag{2.9}$$

2.2.5 Softmax and Loss Layer

The Softmax Layer is used as the last layer on the DNN, after the Dense Layer, and before the Loss computation. The DNN shall output an array of values between 0 and 1 representing the probability of the image of the corresponding class.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.10}$$

The Softmax function takes as input an array z of K real numbers and normalizes it into a probability distribution of K probabilities. The sum of the element is equal to 1 and it tends to maximize the maximum input value. For example, the Softmax output of (1,2,8) is (0.001,0.002,0.997). As you can see, it assigns most of the weight to the greater value, increasing the difference with respect to the other values. To compute the loss out of the Softmax probability we use the cross entropy loss function shown here 2.11, where the Softmax output vector $q(x)$ is multiplied with the expected value $p(x)$ of the inference.

$$L(p, q) = -\sum p(x) \log q(x) \tag{2.11}$$

2.3 Continual learning algorithm

The goal of CL, also known as incremental learning, lifelong learning, or sequential learning, is to gradually learn from different data streams and extend the acquired knowledge [8]. The data streams can be associated with different tasks that the CL system can perform. Incrementally learning all the tasks is an NP-hard problem [9]. Moreover, the major issue of CL is represented by CF, i.e., the ability of the system to perform previously learned tasks degrades over time when new tasks are added [3]. In the literature, several CL methods have been proposed to mitigate CF. CL algorithms can be categorized as regularization-based methods, memory-based methods, and dynamic approaches.

Regularization-based methods apply constraints in the weight update phase to mitigate CF. Weight regularization methods such as Elastic Weight Consolidation (EWC) [10] impose a quadratic penalty to selectively regularize the parameters based on their importance to perform the previous tasks, calculated through the Fisher information matrix. Function regularization methods like the Learning without Forgetting (LwF) [11] employ knowledge distillation to learn the training samples of the new tasks while preserving the knowledge of the previous tasks.

Memory-based approaches, also known as replay-based methods, retrain or finetune the NN jointly using samples from previous tasks and samples from new tasks. The Gradient Episodic Memory (GEM) method [12] constrains the parameter update such that the training loss of each individual previous task does not increase.

Its variant, called A-GEM [13], ensures that the average training loss for all previous tasks does not increase. The Incremental Classifier and Representation Learning (iCaRL) [14] method stores a subset of training samples for each task and jointly minimizes the training loss for new tasks and the distillation loss for the previously learned tasks. Experience Replay (ER) [15] combines training with samples of the new tasks and old samples that are stored in a replay memory. The Maximally Interfered Retrieval (MIR) method [16] selects the samples from the old tasks that would have the largest impact on the forgetting property. The Gradient-based Sample Selection (GSS) strategy [17] maximizes the gradient diversity of the stored sample subset. Greedy Sampler and Dumb Learner (GDumb) approach [18] greedily stores training samples in the memory buffer to maintain a balanced class distribution. The replay data can also be generated at runtime by using the Deep Generative Replay (DGR) method [19], where a generator creates synthetic samples that contain previous tasks knowledge.

Dynamic approaches dynamically increase the DNN architecture to learn features of new tasks. The Continual Neural Dirichlet Process Mixture (CN-DPM) method [20] retains the knowledge of the previous tasks by building a mixture of experts where a new model is trained for a new task, while the existing models for the previous tasks are not modified. The Progressive Segmented Training (PST) method [21] focuses on a single network and, when training on a new task, it divides the parameters into two groups according to their importance to perform that task. The group of important parameters is frozen to preserve the current knowledge, while the other group is saved and can be updated when learning future tasks.

Our architecture supports memory-based approaches due to the simplicity of their hardware implementation, but it can be easily extended to execute other CL algorithms.

2.3.1 DNN Training Accelerators

Various hardware architectures implemented in ASIC or FPGA for accelerating DNN training have been proposed in recent years. The work in [22] is composed of heterogeneous processing tiles to efficiently execute different operations with diverse computational characteristics. The DeepTrain architecture [23] deploys heterogeneous programmable data flows to achieve data reuse during different training operations. The Gist architecture [24] utilizes layer-specific encoding schemes to exploit redundancy in DNN training by storing the feature maps computed during the forward pass and reusing them in the backward pass. The SIGMA accelerator [25] supports irregular sparse workload. To efficiently handle sparsity, the Procrustes accelerator [26] employs a dense tensor dimension for performing arithmetic operations that involve sparse tensors. The LNPU architecture [27] implements fine-grained mixed precision to perform training. The HNPU

architecture [28] supports low-precision training by dynamically configuring the fixed-point representation. The FlexBlock architecture [29] supports multiple block floating-point precisions. The ETA architecture [30] performs training based on the proposed piecewise integer format. The work in [31] trains the DNN through a two-step process that consists of sample collection and policy update for continuous control of the behavior.

The above-mentioned architectures allow the execution of standard training in an efficient manner. However, the execution of CL algorithms requires additional abstraction layers to correctly manage the workload.

Chapter 3

Software Level Implementation of a CNN supporting CL

Different software implementations were developed of the previous CL methods, in order to find the one reaching the higher accuracy. [32] compared them in accuracy, forgetting, and time metrics. The article research came out with the results summaries in Figure 3.1 which is reported here from [32]. The GDumb method gives the best accuracy, at the cost of a higher time. The method suggests that, in the class incremental settings, to diminish forgetting and maintain a high accuracy while learning new tasks, is enough to save a part of the previous training feature in memory and train again the DNN each time new tasks have to be learned. Thus, each time new tasks come and a training cycle starts, we train the network with the current new training feature and the old training features. A Python implementation of the GDumb approach where developed here by the researchers: https://github.com/drimpossible/GDumb.
I developed a CNN Python implementation of the layers previously seen in 2.2: Convolutional layer, Relu, Dense Layer, Batch Normalization Layer, Softmax, and Loss layer. The Python code implements from scratch these functions, in order to test the HW accelerator. The Python code was also implemented to test the data quantization to use in the HW accelerator. To avoid using a 32-bit FP arithmetic (as used generally by classical ML library in Python), we cut the data size to 16 bits using a fixed point format with 4 bits as the integer part and 12 bits as the fractional part.

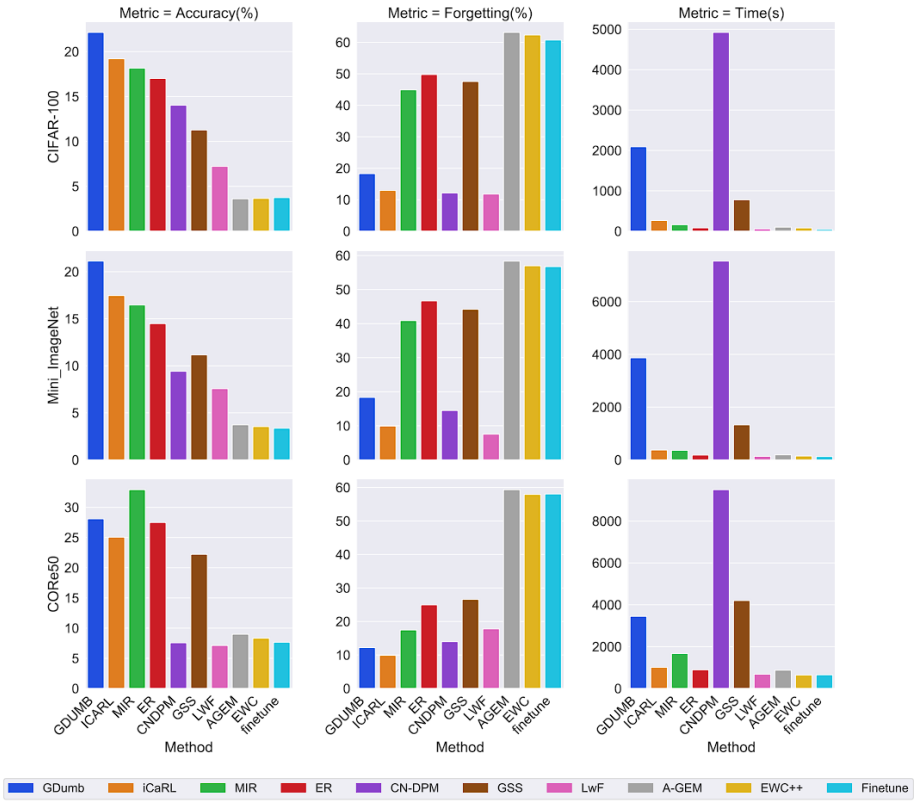

Figure 3.1: Results of different CL methods in 3 metrics: accuracy, forgetting and time.

Chapter 4

Architecture Design

A CNN could be implemented in different architectures. Hence, a little analysis will be carried out, to analyze and motivate which architectural choice could be made and why.

4.1 Early Design Decisions

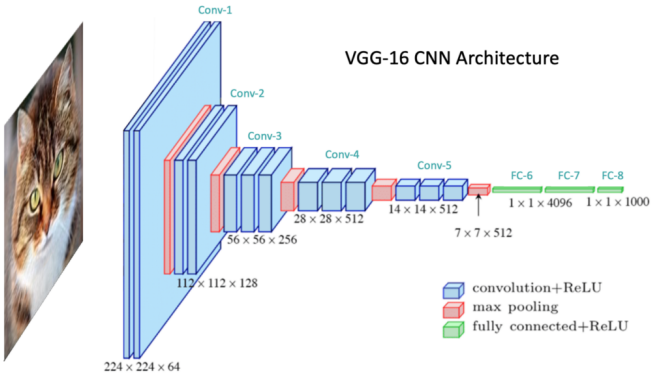

Figure 4.1: Schematic view of a VGG16 CNN

As Figure 4.1 shows, the DNN takes as input a feature x and the first layer computes an output $y = f(x)$, then the second layer computes $y^{'} = f^{'}(y) = f^{'}(f(x))$ and so on so forth. Considering that in CNN, $f(x)$ is often a Convolution operation, the CNN can be simplified to a series of Convolution operations executed in series. However, this convolution operation has to support different feature sizes: as depicted in Figure 4.1, the feature size changes while going through the CNN. To

develop the Convolutional network, I came up with 2 types of network

- **Static**: each software layer corresponds to a physical hardware level. A parametric Convolutional module is instantiated by several times and connected sequentially in order to create a CNN. Each instantiation receives different parameters in order to compute correctly the different feature sizes. In such a way a pipeline operation will be preferred to have a higher HW usage, otherwise only one "layer" will be used, while others will be idle.
- **Dynamic**: Only one Convolutional layer is physically instantiated and it cyclically computes convolution over one feature at a time, adapting the logic to the different sizes. This type of network is more efficient because the single layer is always in use. However, it will involve a more complex control unit. This is because having different sizes means different address spaces, different numbers of iterations, and so on.

Applying the mini-batch principle, the network can be parallelized. As previously seen, we can avoid training one sample at a time. Updating the weights of the model after a batch of samples instead of one has been seen to improve the estimation of the gradient. Also, due to that, we can parallelize the operation of training, summing the derivative of each parameter of each batch and updating the weights at the end of the process. Our processing unit can be :

- **Parallel**: create a copy of the HW that perform training a number of time equal to the batch size. Running on each copy an image of the batch and then sum all the derivatives of the weights.
- **Pipelined**: Using a static network, we can input a feature to the NN as soon as the previous feature concludes the first layer computation.

Different architecture can be used:

- **a parallel static architecture** where each layer is statically instantiated and the number of parallelizations is given by batch size. This is the easiest to implement and the fastest, but it has low HW usage and high area.

$$Area = batchSize \cdot Area_{layer} \cdot N_{layer}.$$

$$Time/sample : \frac{T_{layer} \cdot N_{layer}}{batchSize}$$

- **a parallel dynamic architecture** where each layer is dynamically instantiated. In this architecture, only one “ layer “is instantiated and is dynamically set for the feature size. High HW usage, the same speed as before but higher complexity.

$$Area = batchSize \cdot Area_{layer}.$$

$$Time/sample = \frac{T_{layer} \cdot N_{layer}}{batchSize}$$

- **a parallel static pipelined architecture** where each layer is statically instantiated as in **parallel static architecture**, but in this case, each layer is pipelined, that is when the first feature completes the computation of the first layer and start the computation of the second layer, a second feature is computed in the first layer. Higher throughput, but high area and complexity.

$$Area = batchSize \cdot Area_{layer} \cdot N_{layer}.$$

$$Time/sample = layerDelay \cdot N_{layer} + layerDelay \cdot batchSize$$

The best trade-off between area and speed is the parallel with dynamic instantiation.

As previously said, several accelerator has been developed to execute on-chip training but, as best as our knowledge, none of them support CL methods and catastrophic forgetting policies. Therefore, my accelerator will be composed of 2 parts:

- **A memory**, which applies CL memory-based approach: this memory, paired with a controller, will save a part of training features, usable in the future to re-train the CNN and avoid catastrophic forgetting on previously learned tasks.

- **A trainable CNN** Our NN will implement a Convolutional NN. The RTL system is developed in a general manner, in the sense that it can be configured to implement several layers with different sizes, implementing both forward and feedback computation. Between each layer, a ReLU is present. Due to lack of time, the Dense Layer is not implemented in the RTL, but it will described its implementation in the architecture. Despite I wasn't able to physically implement it, I have considered it in the design of the architecture and I will report how the Dense Layer can be implemented using the same HW used by Convolution. As shown here, 2.6, the Batch normalization layer is hard to implement, and this is only the forward operation. Several studies propose different alternatives to batch normalization:

  - A mixture of normalization of weight, dropout, LR adjustment, and gradient clipping [33].
  - A less computationally expensive normalization algorithm [34].
  - Initialize the NN, normalizing the weights [35].

  I decided to implement gradient clipping [33], for ease of execution and also because it suits well with the data format I chose, which will be explained in the next section.

4.2 Data Quantization

As tested by my Python implementation of a CNN from scratch, a high-precision floating point is not necessary for training DNNs (the results of the tests will be reported in Chapter 5). This is because excessive precision can be also dangerous and slow down the learning process. Fixed point data format is preferred to floating point due to ease in multiplication and addition algorithm: multiplication and addition are quite similar to the integer ones. Floating point, also, will be difficult to be used in a multi-operand adder, because we have to perform exponent adjustment. Also, using fixed point arithmetic instead of floating point we will speed up the critical path, allowing us to use a higher clock. For this reason, a 16-bit fixed-point (4-bit integer + 12-bit fractional data) format is used. The size of the integer part has been chosen according to [33], which suggests using a value clipping of 5 to avoid an Exploding Gradient caused by the lack of a normalization layer.

4.3 Top Level Architecture

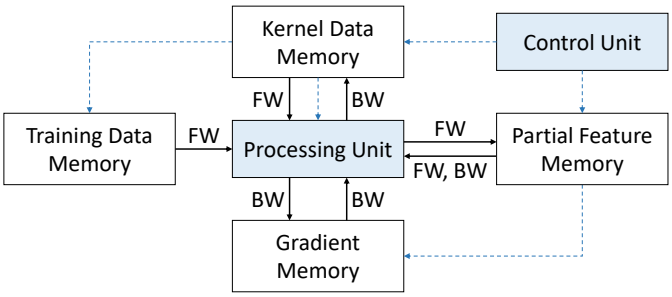

Figure 4.2: Top level architecture. We can see the 4 memories, the PU and the CU.

Generally, a CNN to perform only forward needs to save only its parameters. But in our case, we need to save other data.

4.3.1 Training Data Memory

The GDumb method is a memory method that saves old training samples used to train the network, just in case new classes come to be trained and a new training cycle has to be performed. For this reason, memory is needed to save such samples, which can be updated by adding more samples of new classes and popping samples of old classes so that the number of samples for each class remains equal. To increase data memory throughput I save the 3 channels of RGB image in 3 different

memory. In this way, in one c.c. we can read 3 channels. Also, we will use memory with a port width equal to 128 bits: thus, at each cc, we will read 8 pixels.

4.3.2 Partial Feature Memory

During inference, each layer applies a function $f(k)$ to the input feature, computing an output feature. During backpropagation, the derivative of the output with respect to weights is always a function of the feature. For this reason, for the Convolutional layer, we have to save the input feature during inference, to be used backward. Also in this case, to increase throughput we will use one memory for each feature channel and each memory will have a port width of 128 bit, for the same reason as for training data memory.

4.3.3 Gradient Memory

To save temporarily the gradient of the convolution to be used between 2 layers, a couple of memory has to be used in order to do this. The memory shall be 2 because 1 is not enough: in a multi-channel convolution operation, the pixel we are calculating will overwrite a pixel we will need in further calculus.

4.3.4 Kernel Memory

To save weights to be used during convolution inference and gradient propagation, memory is allocated to kernel values. Using 4D kernel $K_{o,i,j,k}$ in the Convolutional Layer, we use memories of 256 bits where we can save the 9 kernel value. Also here, memory is split according to the input channel and output channel, in order to increase the throughput.

4.4 Processing Unit

Our processing unit, depicted in Figure 4.3, includes 9 parallel multiply and accumulate (MAC) blocks, each of them executing 8 multiplication and 8 addition in parallel. 3 data manager units (Gradient Manager, Kernel Manager, and Feature Manager) are designed to drive the data flow coming from external memory to the MACs.
3 address managers (Gradient Address Manager, Kernel Address Manager, and Feature Address Manager) compute the address of the convolution:

- **The Feature Address Manager** generates the address of the output feature. This address is used in **the Forward of Convolutional operation** to address the output feature pixel to be written. It is also used as a base address to

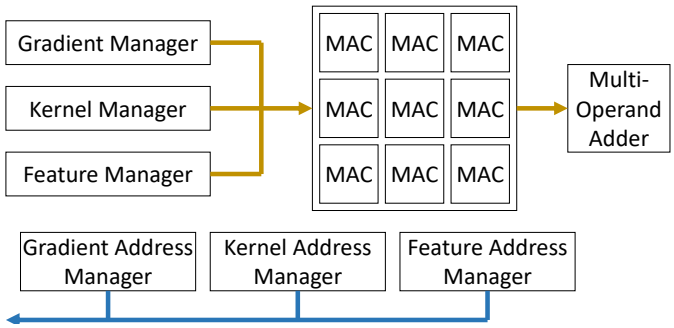

Figure 4.3: Architecture of Processing Unit

compute the address for the input feature. More detail will be given in the subsection of **Forward of convolution operation**. This address is also used in the kernel gradient computation (as explained here 2.5 to address the output feature pixels that have to be read. More detail will be given in the subsection of **Gradient of kernel of convolutional operation**. This address is computed according to the actual size of the feature. It follows the snake movement as explained in the subsection of **Forward of convolution operation**,

- **The Kernel Address Manager** generates the address for kernel memory and is used in **gradient propagation of convolution operation** and **Forward of convolution operation**. It read the 9 kernels for 8 input channels 1 time each convolution.

- **The Gradient Address Manager** computes the gradient address for both feedback computations. Further information will be given in the corresponding subchapter

- **Feature Manager** The Feature Manager rearranges the 3 pixels coming from the CU buffers in order to fill correctly the MAC for Forward computation and Kernel Gradient computation. When I talk about the 3 pixels, I am referring to 8 channels of 3 pixels. For the sake of simplicity, I talk about 3 pixels, but I'm referring to 8 channels of these 3 pixels. In both operations, the manager operates as Figure 4.4 shows The 3 pixels (we use 9 registers of $8 \cdot 16 = 128bits$ where each register contains the values of 8 channels of the pixel) can be loaded in 3 ways: from left, from right, or from bottom. When we are loading and computing an output feature pixel of an even row, in the snake movement I'm proceeding to the right, the 3 pixels are loaded into the right registers. In the case of a spare row, the pixels will be loaded into the left. In case we are moving to the next row, the three pixels are loaded from

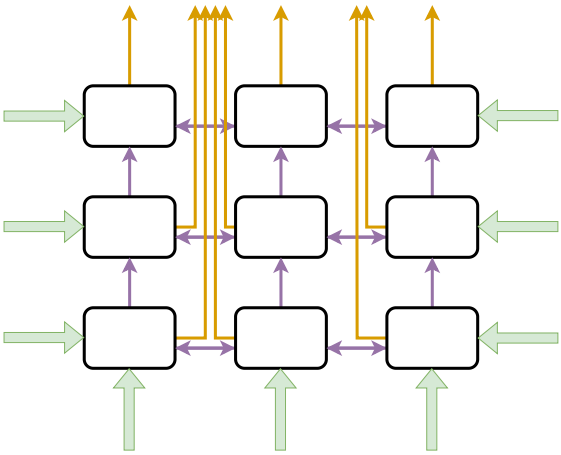

Figure 4.4: Registers manager for forward execution

the bottom registers. These registers are connected among them. When we proceed to the right, the content of the registers is moved to the left. When we are moving to the left, the content or registers are moving to the right. When we move to the bottom, the content of registers moves to the top. All 9 registers are connected to the output, and the register's values are output at each cc.

- **Kernel Manager** The Kernel Registers are used during Forward and Gradient Propagation computation. During Forward computation, the Kernel Register is a regular register, which saves the values of the kernel received at the beginning of a convolution operation for the computation of one channel of the output feature.
  During Gradient Propagation computation, the eq. 2.4 can be seen as the

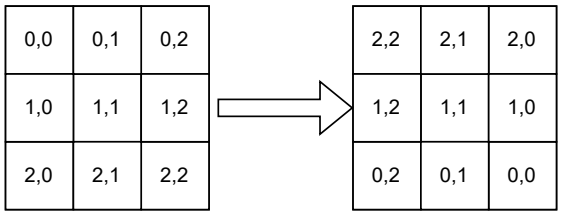

Figure 4.5: The kernel is rotated by 180°

Convolutional operation between the gradient and the kernel matrix rotated by 180°, as Figure 4.5 shows. During this computation, the kernel saved in the registers described before is rotated by 180°.

- **Gradient Manager** The Gradient Manager manages the gradient during

Kernel Gradient computation and Gradient Propagation computation. During Kernel Gradient computation, 8 channels of one pixel are read at each c.c. They are saved in a structure of register, as shown in Figure 4.6. At the beginning of Kernel Gradient computation, the registers are reset and the first pixel is saved in the first register. At each c.c. a new pixel is saved into the register, while the previous is moved to the right. Further description will be given in the section dedicated to Kernel Gradient computation. During

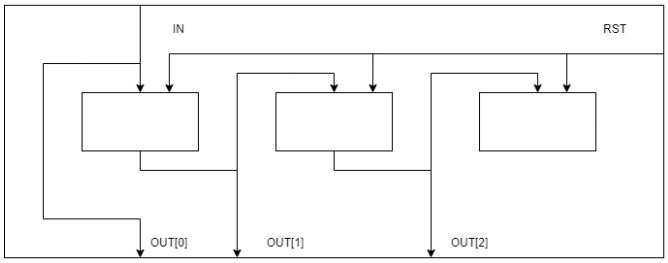

Figure 4.6: The pixel is saved in the first register, At each c.c., a new pixel is loaded and saved in the first register, while the previous one is saved in the next one. Each time we are at column index zero, the 3 registers are reset

Gradient Propagation computation, the Gradient Manager is composed of 9 registers working in the same way as the Feature Register during forward computation and which structure is depicted in Figure 4.4.

4.4.1 MAC

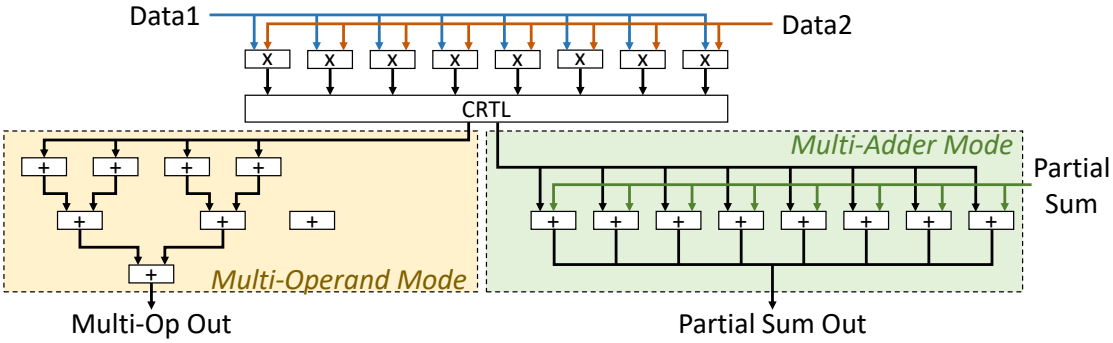

Figure 4.7: Architecture of a MAC. The connections between the adders are dynamics, in order to create a multi-operand adder or a multi-adder structure.

The MAC is composed of 8 adders and 8 multipliers. The 8 adders can be organized in two modes:

- **Multi-Adder mode**: Used during kernel gradient calculation, they sum the 8 multiplication results with 8 input values (PARTIAL SUM in the image). In weight gradient calculation, we parallel compute 8 channels at a time, so each time we have to multiply 8 channels of one input feature pixel with one channel of the input gradient. The results of these 8 multiplications will be summed with the previous multiplication done. More explanations will be provided in the weight gradient part

- **Multi-Operand mode**: During forward and gradient propagation, we compute 8 input channels at a time. In this case, the output is the sum of this 8 input channel (3d convolution) so we configure the 7 adder to operate as a multi operand adder.

To reduce the loss of information, the results of the multiplication are not cut and the adder are 32-bit adder. After the addition phase, the reduction is applied to the 16-bit

4.5 Data Flow

4.5.1 Forward propagation of convolutional operation

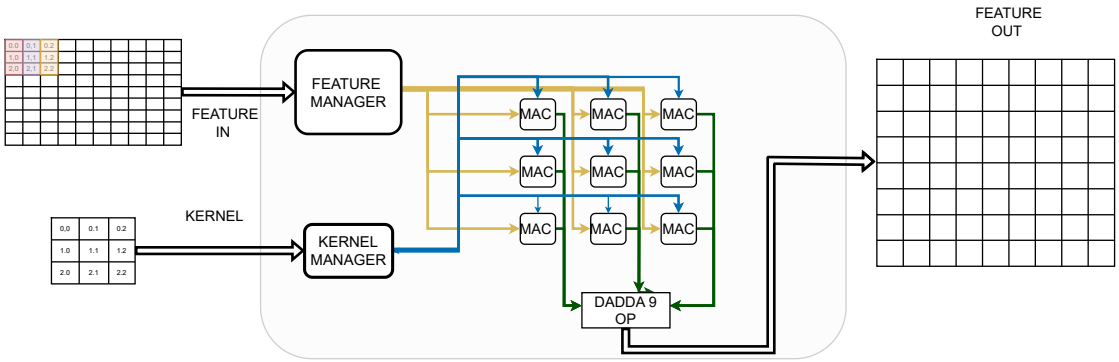

Figure 4.8: Forwand computation detail

In forward operation, a multi-channel input feature has to be 3d convoluted with a 3d kernel to create a single pixel in the new feature.

$$Z(i,j,k) = c(K,V,s)i,j,k = \sum_{l,m,n} [V(l,(j-1)\cdot s+m,(k-1)\cdot s+n)K(i,l,m,n)]$$

To create more channels in the output feature, more 3d kernels are used. Using the MAC earlier explained, we use 9 instances of them to parallel compute a 8x3x3

3d convolution. To output a pixel of the output feature, 8x3x3 input pixels (a 3x3 submatrix for each channel) are convolved with an 8x3x3 matrix of weights (kernels). Most of the input pixels, however, are part of the convolution of 9 output pixels (for each channel out). I decided to maintain part of the previous pixel to be used to the next pixel calculation. This because:

- $F^{out}_{i,j} = F^{in}_{i,j} \cdot k_{i,j} + F^{in}_{i,j+1} \cdot k_{i,j+1} + F^{in}_{i,j+2} \cdot k_{i,j+2} + F^{in}_{i+1,j} \cdot k_{i+1,j} + F^{in}_{i+1,j+1} \cdot k_{i+1,j+1} + F^{in}_{i+1,j+2} \cdot k_{i+1,j+2} + F^{in}_{i+2,j} \cdot k_{i+2,j} + F^{in}_{i+2,j+1} \cdot k_{i+2,j+1} + F^{in}_{i+2,j+2} \cdot k_{i+2,j+2}$

- $F^{out}_{i,j+1} = F^{in}_{i,j+1} \cdot k_{i,j} + F^{in}_{i,j+2} \cdot k_{i,j+1} + F^{in}_{i,j+3} \cdot k_{i,j+2} + F^{in}_{i+1,j+1} \cdot k_{i+1,j} + F^{in}_{i+1,j+2} \cdot k_{i+1,j+1} + F^{in}_{i+1,j+3} \cdot k_{i+1,j+2} + F^{in}_{i+2,j+1} \cdot k_{i+2,j} + F^{in}_{i+2,j+2} \cdot k_{i+2,j+1} + F^{in}_{i+2,j+3} \cdot k_{i+2,j+2}$

The above expression shows how 6 pixels are still used for the next convolution operation.

$$F^{in}_{i,j+1}, F^{in}_{i,j+2}, F^{in}_{i+1,j+1}, F^{in}_{i+1,j+2}, F^{in}_{i+2,j+1}, F^{in}_{i+2,j+2}$$

When we reach the final pixel of a row and move to the next row, we do not restart from column 0. Instead, we start decreasing the column counter. In this way, always 6 pixels are reused. In such ways, for each output pixel, we have to read 3 pixels for 8 channels (16 bits each pixel for 8 channels) and write one value. This is shown in Figure 4.8, where 3 new pixels are loaded each cycle from the input feature into the Feature Register, in which the 3 pixels of the previous 2 reads are saved (at each cycle, the 6 most recent pixels out of 9 stored are saved and 3 new are loaded). These $9 \cdot 8$ (9 values for each input channel) are then sent to the 9 MAC. Each MAC takes 8 channels of one pixel and is multiplied for 8 channels of the one pixel of the kernel. The 9 values are then summed together in a 9 operand Dadda and output the new pixel. If our Input Feature has more than 8 channels (like 16,32, etc...), this operation is repeated and the results of the Dadda operation are accumulated till all the input channels are processed. This is done for the whole output feature.

The address to read the pixels and the kernels and to write out the new feature is generated by the Feature Address Manager and the Kernel Address Manager. This address manager uses the dynamic size given by the CU as a bound of the counter used for column, row, and channel. When the column counter reaches the dynamic size, it will not be zeroed due to snake movement, but it will be maintained and increased by one-row address. This "snake movement" is shown in Figure 4.9. When the end of the matrix is reached, the channel counter is increased.

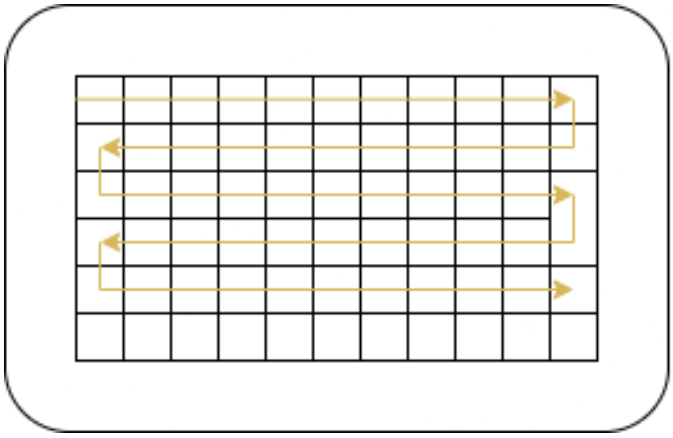

Figure 4.9: Example of snake movement

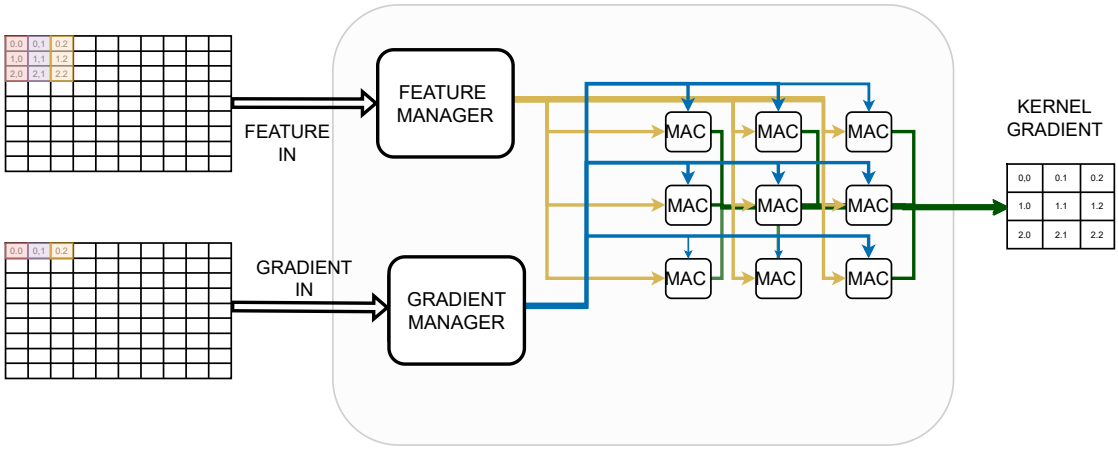

Figure 4.10: Detail of Kernel Gradient computation

4.5.2 Gradient of Kernel of Convolutional operation

To ease the explanations, the kernel gradient formulation is proposed again (see 4.1)

$$g(G,V,s)_i,j,k,l = \frac{\partial}{\partial K_i,j,k,l} J(V,K) = \sum_{m,n} G_{i,m,n} \cdot V_{j,(m-1)\cdot s+k,(n-1)\cdot s+l} \qquad (4.1)$$

where G is the Input Gradient and V is the Input Feature. To compute the gradient of the kernels, a 2D convolution has to be executed. The Input Gradient is convoluted with 9 different submatrices of the Input Feature, where each one is shifted according to the position of the Kernel Gradient it is computing. The input feature is padded with 2 stripes of zeros on the right and bottom border,

having a matrix with sizes increased by 2 in both dimensions. For example, to compute the gradient of kernel 0.0, we will have the submatrix [0:16][0:16](the original non-padded matrix); to compute kernel 1,1 we will have the submatrix [1:17][1:17] (the feature in cutting the first row and column and using one row of padding); and so on. The index of the kernels is also used to point which MAC will be used to compute the gradient, so the gradient of the kernel 0,0 will be computed in MAC 0.0, kernel 1.1 in MAC 1.1, and so on.

$$MAC_{k,l} <= g(G,V,s)_{i,j,k,l} = \frac{\partial}{\partial K_i,j,k,l} J(V,K) = \sum_{m,n} G_{i,m,n} \cdot V_{j,(m-1)\cdot s+k,(n-1)\cdot s+l} \tag{4.2}$$

So, also in this convolution, pixels are shared between convolutions (more than Forward Computation). The extended sum for $MAC_{0,0}$,$MAC_{1,1}$ and $MAC_{2,2}$ is shown in 4.3, 4.4 and 4.5 respectively.

$$\begin{aligned} MAC_{0,0} <= g(G,V,s)_{i,j,0,0} = G_{i,0,0} \cdot V_{j,0,0} + G_{i,0,1} \cdot V_{j,0,1} + \\ ... + G_{i,0,7} \cdot V_{j,0,7} + ... \end{aligned} \tag{4.3}$$

$$\begin{aligned} MAC_{1,1} <= g(G,V,s)_{i,j,1,1} = G_{i,0,0} \cdot V_{j,1,1} + G_{i,0,1} \cdot V_{j,1,2} + ... \\ + G_{i,0,7} \cdot V_{j,1,8} + ... + G_{i,1,0} \cdot V_{j,2,1} + ... \end{aligned} \tag{4.4}$$

$$\begin{aligned} MAC_{2,2} <= g(G,V,s)_{i,j,2,2} = G_{i,0,0} \cdot V_{j,2,2} + G_{i,0,1} \cdot V_{j,2,3} + ... \\ + G_{i,0,7} \cdot V_{j,2,9} + ... + G_{i,1,0} \cdot V_{j,3,2} + ... \end{aligned} \tag{4.5}$$

I would like to recall that at each c.c. we can read 8 pixels of 8 channels of the input Gradient $G_{[i:i+7,m,n:n+8]}$ and 8 pixels of 8 channels of the input Feature $V_{[j:j+7,m,n:n+8]}$

To compute 4.2 for each MAC, we need to read at each c.c. 3 pixels, as 4.6 shown. In order to do so, prefetching is executed.

$$V_{[j:j+7,m,n:n+8]} V_{[j:j+7,m+1,n:n+8]} V_{[j:j+7,m+2,n:n+8]} \tag{4.6}$$

In this way, we can compute in parallel the kernel of row indexes 0,1 and 2. The column shift, instead, is done by delaying the pixels of the gradient. This can be seen more clearly if we write the computation of Kernel Gradient 0.2, as shown in 4.7.

$$\begin{aligned} MAC_{0,2} <= g(G,V,s)_{i,j,0,2} = 0 \cdot V_{j,0,0} + 0 \cdot V_{j,0,1} + G_{i,0,0} \cdot V_{j,0,2} + G_{i,0,1} \cdot V_{j,0,3} + \\ ... + G_{i,0,6} \cdot V_{j,0,8} + G_{i,0,7} \cdot V_{j,0,9} + ... + G_{i,1,0} \cdot V_{j,1,2} + ... \end{aligned} \tag{4.7}$$

The equation shows that the Input Feature is multiplied with a delayed value of the gradient, where the delay is equal to the kernel columns. This delay is done by the Gradient Manager described in the Processing Unit section.

The computation is repeated in order to compute all the Kernel Gradient. The Input Feature V, instead, does not need to be prefetched, because we have to use only one pixel each c.c.

4.5.3 Gradient Propagation of Convolution operation

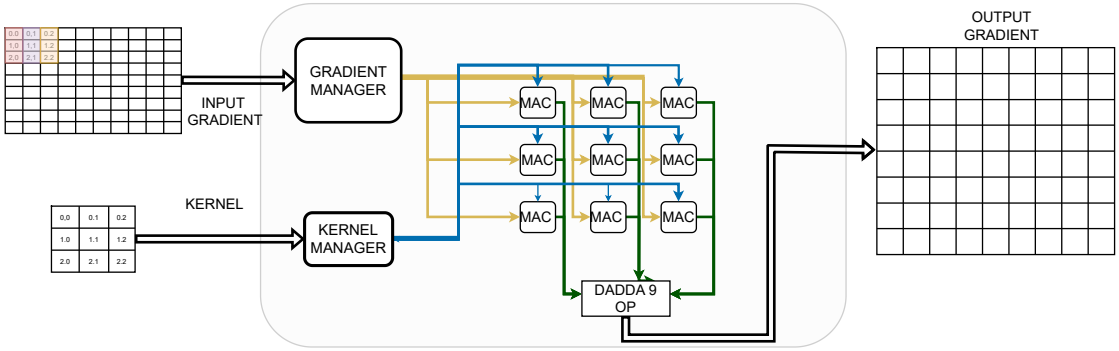

Figure 4.11: Gradient Propagation computation

To compute the propagation of the gradient, we need to execute a 3D convolution between a multichannel 3D kernel and a multichannel 3D Gradient propagated from the previous layer, to create a single-channel gradient. To compute a multichannel gradient, more 3D kernels are used to execute convolution with the same input gradient. As Figure 4.11, the data flow is equal to the Forward propagation of the Convolutional operation. The Gradient Manager receives 3 pixels each c.c. from the CU buffers and generates the Convolution Matrix to be convoluted with the kernel.

4.5.4 Dense layer computation

The Dense Layer computation is done using the same HW used during convolution.

- **Forward computation**: The Forward is implemented as a matrix multiplication between the flatted pixels of the last convolutional layer Output Feature and a 2D matrix. The Weight Matrix has a size (m,n) where m is the number of pixels of Input Feature and n is the number of Output Value. The output value is equal to the number of inference classes. This number, due to CL configuration, is not static and changes during operation. Anyway, the output of the Forward Computation is implemented as 4.8.

$$y_n = \sum_{i=0}^{m} I_i \cdot W_{i,n} \tag{4.8}$$

  This can be re-write as 4.9 considering the input value is not flattened, so if it is still a 3d matrix with sizes (I, J, K)

$$y_n = \sum_{i,j,k=0}^{I,J,K} I_{i,j,k} \cdot W_{i,j,k,n} \tag{4.9}$$

  So in this way, we can use the same logic used during the Forward computation for convolution. In this case, 8 pixels of 8 channels of the Input Feature and 8 weights of 8 channels are read each c.c and sent to 8 of 9 MAC. All 64 results are then summed together and saved into the same register used to save partial sum during Kernel Gradient computation. This is done for the whole feature and it is repeated n time, where n is the number of classes. In this way, we can dynamically set the number of iterations we have to do.

- **Gradient Propagation computation** The Gradient Propagation is equal to the matrix multiplication between input gradient propagation and the transpose of the weights, as shown in 4.10

$$dX_i = \sum_{n=0}^{N} dY_n \cdot W_{n,i}^T \tag{4.10}$$

  In this case, the optimization is a little bit tricky. Due to the fact the dimensionality of dY is dynamic due to CL configuration and is not a power of 2, we cannot reach a 100 % HW utilization. We propose to execute inside a single MAC the computation for each single pixel of the Gradient Propagation, using the register and the partial sum logic to iterate. Using 9 MAC, we can

compute 9 pixels in N/8 c.c., where N is the number of classes and the whole computation in (I/9)(N/8), as shown in4.11.

$$MAC_{0,0} <= dX_0 = \sum_{n=0}^{7} dY_n \cdot W_{n,0}^T \tag{4.11}$$

The values will be saved in the register used for partial sum and summed in the next addition, till the end of the vector. The vectors will be padded with 0 to match.

- **Weight Gradient computation** The derivative over the weight is the matrix multiplication of the input feature I used during forward computation and gradient propagation coming from loss computation dY, as shown in 4.12

$$dW_{i,n} = I_i \cdot dY_n \tag{4.12}$$

Using matrix notation, both are row matrix, so the matrix product produce a matrix. This means the whole matrix I is multiplied with the same value dY_n. This calculus can be executed in our MAC: a 64 parallel pixel read is executed (8 pixels for 8 channels, as in forward) and multiplied with one pixel of dY. In this case, neither addition inside a single MAC is executed and outside the 9 MAC. This is repeated for each pixel of dY, so also in this case, due to the dynamic size of Gradient Propagation due to class incremental.

4.6 Control Unit

The control unit manages the sizes of the feature and the data flow between the memories and the PU. As seen in the previous subsection, data timing is quite important:

- **Feature prefetch**: At each c.c., 3 new input feature pixels have to be sent to the feature manager, in order to build the matrix that will be sent to MACs. To have them ready without stalling the operation, a prefetch operation is used, where 3 blocks of 128 bits are read from the memory. The prefetch operation starts when the sixth pixel of the block is convolved.

  In Figure 4.12, the prefetching operating for even rows is shown. When the sixth pixel is convolved, it starts to read the 3 rows of pixels. To be noticed, that here we are talking about the pixels that are loaded into PU to be managed by the Feature Manager. As explained in 4.5.1, when we load into PU the pixel of index j+5, we are computing the convolution for the output pixel j+3. Figure 4.13 instead, shows in case of the spare row address. The last prefetching of a row is shown in Figure 4.14 and Figure 4.15. Here, due to snake movement, only one new read has to be performed.

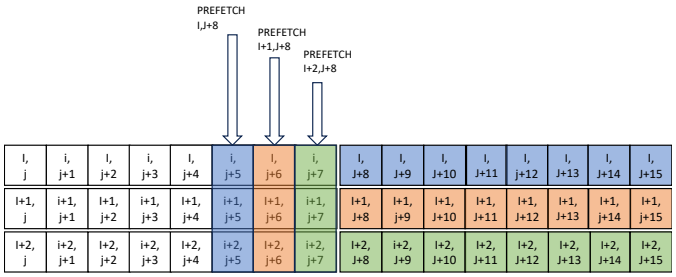

Figure 4.12: When the column j+5 is elaborated, the prefetching of the row i is performed; When the column j+6 is elaborated, the row i+1 is prefetched; When the column j+7 is elaborated, the row i+2 is prefetched

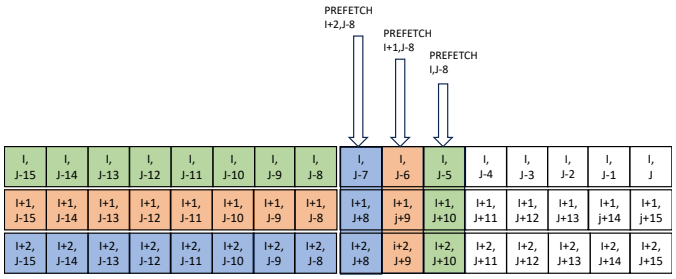

Figure 4.13: Prefetch operation during spare row address.

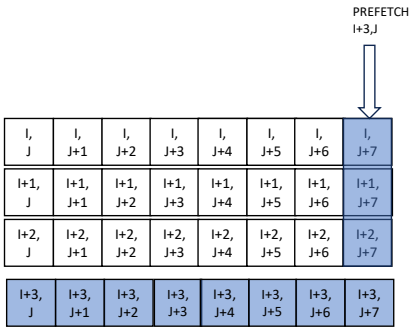

Figure 4.14: Prefetch operation on the last block when row address is even

- **Kernel Prefetch**: Each time an 8-channel feature is convoluted, the new 8 channels of $3x3$ kernel have to be loaded. In this case, parallel reads are executed over multiple channels (at each c.c., multiple channels can be read,

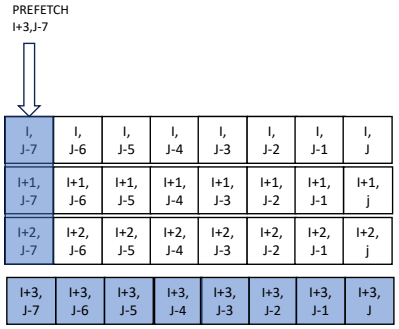

Figure 4.15: Prefetch operation on the last block when the row address is spare. When the prefetch operation is performed on the last column, only one read is performed of the row i+3, because the rows i+1 and i+2 are already in the buffer

due to the memories configuration).In any case, to read 9 kernel values, 9 c.c. are needed. The kernel is used during forward computation and gradient propagation computation. The kernel prefetching happens when the input feature (during forward computation) or the input gradient (during gradient propagation computation) is 9 c.c. to the end. As during feature prefetch, at each c.c. a kernel is read and when the new feature computations start, the kernel is loaded into PU.

- **Gradient Prefetch** The input gradient is read in both backward operations: Gradient propagation and Kernel Gradient computation. In the first case, the gradient is convolved with kernel values rotated by 180°. In this case, prefetching is used, because the computation is similar to the forward one. Instead, during kernel propagation, no prefetching is needed.

Finally, the CU manages the feature, kernel, and gradient write phase. Due to port width, feature and gradient use a 128-bit buffer to temporarily save the output of the computation. Each 8 c.c., a write operation is performed. To reduce buffer utilization, feature and gradient reuse the same buffer during Read operation, because we do not need to prefetch both at the same time.

4.7 GDumb control unit

The GDumb method [36] works on which training samples have to be saved inside the training method. To avoid catastrophic forgetting and imbalanced learning, the GDumb method proposes to save an equal number of training samples for each class.

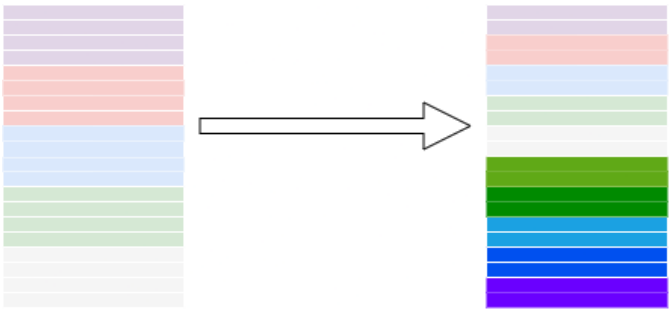

Figure 4.16: Gdumb method. When the number of class increase, the number of training sample for each class decrease

As shown in Figure 4.16, when new classes have to be learned, the new training samples are saved in memory to the detriment of training samples of old classes. In our accelerator, a controller has been designed so that, each time new classes are added, a rewrite cycle is performed. It is preferred a rewrite instead of just adding the new feature into the position of a feature that shall be deleted because it works also in case the class increment is not a doubling of class: as the image above suggests, if we want to increase from 5 to 10 classes (but also from 10 to 20, 50 to 100, etc) we can just write the new training samples in the upper (or also lower) half of the memory location of one class. This doesn't work if we want to update from 10 to 15 for example. In this case, we first rewrite the feature that can be saved, and then a part of the new Training Feature a saved, according to the space available for each image.

Chapter 5

Simulation and Syntesis

5.1 Hardware validation

The purpose of the Python software implementation of the model is to validate the HW hardware. To validate the software model, two TensorFlow model has been developed. The TensorFlow models and my models have the following structures:

- **Single Layer (SL)** A convolutional layer from 32x32x3 to 16x16x16, activated with a ReLU,
- **Double Layer (DL)** Two convolutional layer from 32x32x3 to 16x16x16, activated with ReLU, and a convolutional layer of 16x16x16 to 8x8x32, activated with a ReLU

Both models have a dense and softmax layer a the end. The models have been tested on CIFAR10, with no Class Incremental settings, so the Dense and Softwamx layers operate on a vector of cardinality equal to 10. Also, my software implementation operates on a 16-bit fixed point data type, to simulate the HW data. Due to the slowness of Python, I was able to run the simulation only till epoch 10, but it was enough to make both models reach the maximum accuracy. In Table 5.1 the results are shown. My Python model is able to learn, although it is not able to reach the same base accuracy as TF.

The complete TinyCL architecture has been described in the RTL level using the SystemVerilog language. We synthesize the architecture in a 65 nm CMOS technology node with the ASIC design flow using the Synopsys Design Compiler tool. A Python testbench environment has been developed in order to test the RTL. The results of forward, gradient propagation, and kernel gradient computation have been validated by comparing the results of the 3 operations(Forward, Gradient Propagation and Gradient of the Kernel) with the results of the computation executed in Python. In convolution forward computation, the GDumb memory and

Architecture	dataset	training epoch	Accuracy
SL (mine)	CIFAR10	10	**38%**
SL (TF)	CIFAR10	10	40%
DL (mine)	CIFAR10	10	**49%**
DL (TF)	CIFAR10	10	54%

Table 5.1: Comparision between my Python code implementing a convolutional NN from scratch and the same NN of TF.

the kernel memory are filled by the testbench with the same values used by Python code, and the same is done with gradient during gradient and kernel gradient computation.

5.2 Syntesis timing results

Our architecture has a critical path starting from a memory (Gdumb Memory, Feature Memory, Kernel Memory, or Gradient Memory) and it is concluded to another memory. At each c.c., in general :

- **read**: We read a value from the prefetch buffer. The prefetch operation is done previously, in parallel with the ongoing operations.
- **MAC operation** : a MAC operation is executed. A MAC operation consists of 72 multiplications (8 for each of the 9 Macs) and 72 additions (8 for each of the 9 Macs). For FW and DX operation, also a multi-operand addition is present. The worst case is present when FW and DX as to be computed where the time for the critical path T_{cp} is :

$$T_{cp} = T_{mult} + 3 \cdot T_{add} + T_{dadda}$$

- **write** The result is write. The write operation is executed in the write buffer.

Using the Synopsis design compiler and a synthesis script, the Synopsis tool reports that my accelerator has a critical path T_{cp} equal to 3.87 ns. For this reason and choosing a bigger and standard clock, we set the clock frequency to 250 MHz. I test the timing speed using the DL model. The TensorFlow model is accelerated using the NVIDIA Tesla P100 GPU. To compute the Forward computation and the Gradient Propagation of the first Convolutional layer my accelerator takes 16*16*16 = 4096 c.c. To compute the Forward computation and the Gradient Propagation of the second Convolutional layer my accelerator takes 2*8*8*32 = 4096 c.c. To compute the Kernel Gradient propagation of the first layer my accelerator takes

16*16*16 = 4096 c.c., while the second layer took 32*8*8*2 = 4096 c.c. The Dense layer has to multiply 32*8*8 = 2042 pixels with 2042 weight for the 10 classes, but we execute 64 multiplication each cc, so the Forward operation of the Dense layer takes (2042/64)*10 = 320 cc To compute the Gradient Propagation, we will have

$$dX_i = \sum_{n=0}^{10} dY_n * W_{n,i}^T$$

As previously said, the sum over n is "decomposed" inside one MAC and each MAC computes one pixel of the Gradient propagation. So to compute one pixel each MAC it takes 2 cc, so 9 pixels are computed each 2 cc. To compute 2042 Gradient propagation matrix pixels it takes 2042*2/9 = 454 c.c. Finally, to compute the Gradient of the Kernel, 2042 pixels are parallelly multiplied with 10 Gradient propagation pixels, so it takes 2042/64*10 = 320 cc, as FW.

So the total requested time for one image training is

$$T = T_{conv1}^{FW} + T_{conv2}^{FW} + T_{dense}^{FW} + T_{dense}^{DK} + T_{dense}^{DX} + T_{conv2}^{DK} + T_{conv1}^{DX} + T_{conv1}^{DK} + =$$

$$4096 + 4096 + 320 + 454 + 320 + 4096 + 4096 + 4096 = 21574cc$$

The Model where tested in the CL environment, using a training memory of 1000 samples To execute a 10 epoch training it takes 21574 * 1000 * 10 = 215740000 cc Considering a clock of 250 Mhz, (so a period of 4ns), our model takes 0.862 s. The TensorFlow model trained using the NVIDIA Tesla P100 GPU took on average 4 s each epoch, taking 40s to execute the complete training, reaching a speed up of 46x. This high speed-up is possible because we were using a batch size of 1, and the GPU accelerate the ML training executing in parallel the images of a batch.

Chapter 6

Conclusion

In this work, I show how we can speed up CNNs by accelerating and optimizing the Convolution layer operation in a CL scenario. I show how we can reduce Data memory access by exploiting data reuse. This can reduce data access from 9 to 3 c.c. each. This is no longer true if the convolution operation has a stride of 2: in this case, 2 consecutive convolution has only 3 shared pixels during the Forward operation (and this will affects also the feedback).
The prefetching and the memories port-wide let the system run without stalling, despite more than one data location is needed. This architecture leads to a great speed-up of CNNs, thanks to convolutional layer acceleration, compared to NVIDIA Tesla P100.
A preliminary Architectural design, shown in Section 4.5.4 shows how the Dense Layer fits well and can be computed using the 9 MACs. The architecture shall also include a module that implements the last Softmax layer and the Loss computation.
A further optimization can be given by memory caching. Actually, the memories were designed as SRAM during the various tests. But with the rising of the number of layers, the SRAM can drastically rise, while the PU still compute data coming from one layer at a time. A solution is to save all the Kernel, Feature, Training Samples, and Gradient in an off-chip RAM and to cache it only when needed.

Bibliography

[1] Pudi Dhilleswararao, Srinivas Boppu, M. Sabarimalai Manikandan, and Linga Reddy Cenkeramaddi. «Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey». In: *IEEE Access* (2022) (cit. on p. 1).

[2] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. «A Comprehensive Survey of Continual Learning: Theory, Method and Application». In: *CoRR* abs/2302.00487 (2023) (cit. on p. 1).

[3] Michael McCloskey and Neal J Cohen. «Catastrophic interference in connectionist networks: The sequential learning problem». In: *Psychology of learning and motivation*. 1989 (cit. on pp. 1, 7).

[4] Yiran Chen et al. «A survey of accelerator architectures for deep neural networks». In: *Engineering* (2020) (cit. on p. 2).

[5] Maurizio Capra et al. «An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks». In: *Future Internet* (2020) (cit. on p. 2).

[6] Jinsu Lee and Hoi-Jun Yoo. «An overview of energy-efficient hardware accelerators for on-device deep-neural-network training». In: *IEEE OJ-SSCS* (2021) (cit. on p. 2).

[7] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *International conference on machine learning*. pmlr. 2015, pp. 448–456 (cit. on p. 6).

[8] Sebastian Thrun and Tom M. Mitchell. «Lifelong robot learning». In: *RAS* (1995) (cit. on p. 7).

[9] Jeremias Knoblauch, Hisham Husain, and Tom Diethe. «Optimal Continual Learning has Perfect Memory and is NP-hard». In: *ICML*. 2020 (cit. on p. 7).

[10] James Kirkpatrick et al. «Overcoming catastrophic forgetting in neural networks». In: *CoRR* abs/1612.00796 (2016) (cit. on p. 7).

[11] Zhizhong Li and Derek Hoiem. «Learning without Forgetting». In: *IEEE TPAMI* (2018) (cit. on p. 7).

[12] David Lopez-Paz and Marc'Aurelio Ranzato. «Gradient Episodic Memory for Continual Learning». In: *NeurIPS*. 2017 (cit. on p. 7).

[13] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. «Efficient Lifelong Learning with A-GEM». In: *ICLR*. 2019 (cit. on p. 8).

[14] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. «iCaRL: Incremental Classifier and Representation Learning». In: *CVPR*. 2017 (cit. on p. 8).

[15] Tyler L. Hayes, Nathan D. Cahill, and Christopher Kanan. «Memory Efficient Experience Replay for Streaming Learning». In: *ICRA*. 2019 (cit. on p. 8).

[16] Rahaf Aljundi et al. «Online Continual Learning with Maximal Interfered Retrieval». In: *NeurIPS*. 2019 (cit. on p. 8).

[17] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. «Gradient based sample selection for online continual learning». In: *NeurIPS*. 2019 (cit. on p. 8).

[18] Ameya Prabhu, Philip H. S. Torr, and Puneet K. Dokania. «GDumb: A Simple Approach that Questions Our Progress in Continual Learning». In: *ECCV*. 2020 (cit. on p. 8).

[19] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. «Continual Learning with Deep Generative Replay». In: *NeurIPS*. 2017 (cit. on p. 8).

[20] Soochan Lee, Junsoo Ha, Dongsu Zhang, and Gunhee Kim. «A Neural Dirichlet Process Mixture Model for Task-Free Continual Learning». In: *ICLR*. 2020 (cit. on p. 8).

[21] Xiaocong Du, Gouranga Charan, Frank Liu, and Yu Cao. «Single-Net Continual Learning with Progressive Segmented Training». In: *ICMLA*. 2019 (cit. on p. 8).

[22] Swagath Venkataramani et al. «ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks». In: *ISCA*. 2017 (cit. on p. 8).

[23] Duckhwan Kim, Taesik Na, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. «DeepTrain: A Programmable Embedded Platform for Training Deep Neural Networks». In: *IEEE TCAD* (2018) (cit. on p. 8).

[24] Animesh Jain et al. «Gist: Efficient Data Encoding for Deep Neural Network Training». In: *ISCA*. 2018 (cit. on p. 8).

[25] Eric Qin et al. «SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training». In: *HPCA*. 2020 (cit. on p. 8).

[26] Dingqing Yang et al. «Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training». In: *MICRO*. 2020 (cit. on p. 8).

[27] Jinsu Lee et al. «7.7 LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16». In: *ISSCC*. 2019 (cit. on p. 8).

[28] Donghyeon Han et al. «HNPU: An Adaptive DNN Training Processor Utilizing Stochastic Dynamic Fixed-Point and Active Bit-Precision Searching». In: *IEEE JSSC* (2021) (cit. on p. 9).

[29] Seock-Hwan Noh et al. «FlexBlock: A Flexible DNN Training Accelerator With Multi-Mode Block Floating Point Support». In: *IEEE Trans. Computers* (2023) (cit. on p. 9).

[30] Jinming Lu, Chao Ni, and Zhongfeng Wang. «ETA: An Efficient Training Accelerator for DNNs Based on Hardware-Algorithm Co-Optimization». In: *IEEE TNNLS* (2023) (cit. on p. 9).

[31] Changhyeon Kim et al. «A 2.1TFLOPS/W Mobile Deep RL Accelerator with Transposable PE Array and Experience Compression». In: *ISSCC*. 2019 (cit. on p. 9).

[32] Zheda Mai, Ruiwen Li, Jihwan Jeong, David Quispe, Hyunwoo Kim, and Scott Sanner. «Online continual learning in image classification: An empirical survey». In: *Neurocomputing* 469 (2022), pp. 28–51 (cit. on p. 10).

[33] Divya Gaur, Joachim Folz, and Andreas Dengel. «Training deep neural networks without batch normalization». In: *CoRR abs/2008.07970* (2020) (cit. on pp. 14, 15).

[34] Shuang Wu, Guoqi Li, Lei Deng, Liu Liu, Dong Wu, Yuan Xie, and Luping Shi. «$L1$ -Norm Batch Normalization for Efficient Training of Deep Neural Networks». In: *IEEE Transactions on Neural Networks and Learning Systems* 30.7 (2019), pp. 2043–2051. DOI: 10.1109/TNNLS.2018.2876179 (cit. on p. 14).

[35] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. «Fixup initialization: Residual learning without normalization». In: *arXiv preprint arXiv:1901.09321* (2019) (cit. on p. 14).

[36] Ameya Prabhu, Philip HS Torr, and Puneet K Dokania. «Gdumb: A simple approach that questions our progress in continual learning». In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*. Springer. 2020, pp. 524–540 (cit. on p. 28).