Processing and learning deep neural networks on chip

Ghouthi Boukli Hacene

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THESE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPERIEURE MINES-TELECOM ATLANTIQUE
BRETAGNE PAYS DE LA LOIRE - IMT ATLANTIQUE
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité :Electronique, Informatique, Signal, Image, Vision
Par

Ghouthi BOUKLI HACENE

Processing and Learning Deep Neural Networks on Chip

Thèse présentée et soutenue à Brest, le 03/10/2019
Unité de recherche : Lab-STICC, UMR CNRS 6285
Thèse N° : 2019IMTA0153

Rapporteurs avant soutenance :

Julie GROLLIER	Directrice de recherche, CNRS/Thales
Warren GROSS	Professeur, McGill

Composition du Jury :

Rapporteurs :	Julie GROLLIER	Directrice de recherche, CNRS/Thales
	Warren GROSS	Professeur, McGill
Examinateurs :	Hervé JEGOU	Chercheur, Facebook AI Research
	Yoshua BENGIO	Professeur, Université de Montréal
	Vincent GRIPON	Chercheur permanent, IMT Atlantique
Dir. de thèse :	Michel JEZEQUEL	Professeur, IMT Atlantique

Invité(s)

Nicolas FARRUGIA	Maitre de conférence, IMT Atlantique
Matthieu ARZEL	Maitre de conférence, IMT Atlantique

Table of Contents

Résumé

Introduction

L'apprentissage machine fait référence à un domaine de l'informatique dans lequel le principe est d'apprendre à partir d'exemples, d'expériences et/ou d'interactions. Au lieu d'être explicitement codés pour exécuter une tâche spécifique, les algorithmes développés dans ce domaine sont donc en mesure d'acquérir leurs fonctionnalités d'une manière qui est sans aucun doute beaucoup plus proche de la façon dont l'homme apprend, de sorte que l'apprentissage machine est un sous-domaine de l'intelligence artificielle. De nombreuses raisons justifient et motivent l'utilisation de l'apprentissage machine. Par exemple, dans certains cas, il n'y a pas de solution connue au problème, comme pour la classification des images. Dans d'autres cas, les solutions connues sont trop coûteuses sur le plan informatique, et l'apprentissage machine apporte des compromis intéressants entre la justesse et la vitesse des algorithmes.

L'apprentissage machine n'est pas une méthode en soi, mais plutôt un ensemble de méthodes telles que les machines à vecteurs de support (SVM), les forêts aléatoires d'arbres de décision, et l'apprentissage de réseaux de neurones profonds. Ces derniers ont suscité le plus d'intérêt au cours de ces dernières années. L'apprentissage profond est basé sur un algorithme inspiré du cerveau appelé réseau de neurones artificiel, dans lequel les neurones sont connectés et échangent des informations entre eux.

Grâce à l'intérêt que ce domaine a suscité au cours des deux dernières décennies, l'apprentissage machine en général et l'apprentissage profond en particulier sont devenus l'état de l'art dans de nombreux domaines comme la vision par ordinateur, la reconnaissance vocale, le traitement du langage naturel et même les jeux, dépassant ainsi les capacités humaines pour certains tâches. Cependant, pour atteindre des performances de l'état de l'art, l'apprentissage profond utilise une grande quantité de ressources, y compris de la mémoire pour stocker les modèles et les données, et des calculs pour traiter les différentes données, ce qui conduit à une grande consommation d'énergie, et

à un temps de calcul considérable. De tels besoins peuvent rapidement devenir une limitation qui limite les domaines d'application d'apprentissage profond. La mémoire, la puissance de calcul et la consommation énergétique représentent des ressources clés que les méthodes d'apprentissage profond récemment introduites visent à préserver, et qui soulèvent des défis scientifiques, techniques et même sociétaux.

Dans ce manuscrit, nous cherchons à réduire la mémoire et la complexité (ou le nombre d'opérations) de l'apprentissage profond, car ce sont les deux principales limitations qui engendrent les différents défis, et nous abordons le problème de la prédiction et de l'apprentissage sur puce. Nous passons en revue et introduisons certaines méthodes qui visent à réduire la taille et la complexité des modèles d'apprentissage profond, ainsi que d'autres méthodes permettant un apprentissage incrémental très performant, dans lequel les données sont apprises au fur et à mesure.

Réduction de la complexité de l'inférence

Afin de faciliter l'implantation des réseaux de neurones sur des systèmes embarqués à faible ressources, certains travaux ont été proposés afin de réduire l'utilisation de la mémoire et/ou le nombre d'opérations. Les principales approches sont les suivantes. Certains travaux visent à utiliser des approches de haut niveau et proposent d'utiliser des techniques d'élagage pour réduire le nombre de connexions dans les architectures de réseaux de neurones [56, 62, 32, 107], ou de factorisation pour fusionner plusieurs parties des architectures [28, 105]. D'autres approches utilisent des architectures de réseaux de neurones légers [40], des convolutions groupées [37, 86], ou remplacent la convolution par un décalage de l'entrée suivit d'une multiplication [104, 44, 23]. Nous avons introduit durant la thèse une nouvelle méthode appelé Shift Attention Layer (SAL) [26], une méthode d'élagage, qui pendant la phase d'apprentissage choisi de ne garder qu'un seul poids par noyau de convolution, et donc remplace la convolution par une multiplication. SAL surpasse les autres méthodes de compression de l'état de l'art en terme de justesse, de nombre de paramètres et nombre de calculs. Dans d'autres travaux, les auteurs proposent d'utiliser des approches de bas niveau telles que la quantification des valeurs de poids et/ou d'activation sur n bits ($n < 32$) [101, 67, 118], jusqu'aux cas extrêmes où elles deviennent ternaires [57] (habituellement -1,0, +1) ou même binaires (habituellement -1 ou $+1$) [11]. Durant nos travaux de thèse, nous avons également pensé à une méthode très bas niveau pour réduire la consommation d'énergie du réseau de neurones et qui consiste à tout simplement réduire la tension d'alimentation du système embarqué [27]. Afin de réduire au mieux l'énergie de consommation tout en gardant une justesse accept-

able, nous avons proposé d'appliquer les même conditions, a savoir réduire la tension d'alimentation durant la phase d'apprentissage, et donc adapter le réseau de neurones a de telles conditions.

Dans ce manuscrit nous passons en revue différentes méthodes de compression, et nous introduisons une comparaison critique de ces méthodes. En particulier, dans la littérature et dans la plupart des méthodes discutées dans ce manuscrit, les auteurs comparent la justesse et le nombre de paramètres de leurs méthodes avec une référence choisie a priori. Cependant, un tel processus ne donne que deux points qui ne peuvent être utilisés pour effectuer une comparaison équitable. Une bonne comparaison consisterait donc à comparer la justesse pour le même nombre de paramètres et vice versa.

Les méthodes de compression peuvent être efficaces pour réduire la mémoire et le nombre d'opérations nécessaire pour traiter une donnée à travers le réseau de neurones et (par exemple) la classifier. Cependant, de telles méthodes ne sont pas adaptées pour être utilisées durant la phase d'apprentissage, qui est une phase très complexe et très coûteuse en ressource, et donc ne peuvent pas réduire sa complexité.

Apprentissage incrémental

Afin de répondre aux problèmes liés à la phase d'apprentissage, et la rendre moins coûteuse en terme de mémoire et d'opérations, nous proposons dans ce document d'étudier les solutions incrémentales, permettant d'apprendre au fur et à mesure qu'on fournit de nouvelles données. Il s'agit d'une méthode permettant à un modèle d'apprendre les données de façon séquentielle, utilisant à chaque étape des sous-ensembles de la base de données. Plus précisément, une approche d'apprentissage incrémental peut être définie par [77, 78] : a) la capacité d'apprendre des informations supplémentaires à partir de nouvelles données (incrément par les exemples), b) l'absence du besoin de stocker ou de réutiliser les données originales qui ont servi à entraîner les classifieurs (afin de limiter l'occupation mémoire), c) la préservation des connaissances préalablement acquises (éviter l'oubli catastrophique) et d) la capacité de gérer de nouvelles catégories qui peuvent être introduites avec de nouvelles données (incrément par les catégories). Donc, dans le contexte des systèmes embarqués, la notion d'apprentissage incrémental prend tout son sens, car elle permet de réduire la complexité d'apprentissage en apprenant qu'un exemple à la fois, et de limiter la mémoire car elle ne nécessite pas de stocker en mémoire toutes la base de données d'apprentissage.

Certaines méthodes d'apprentissage incrémental ont été proposées dans la littérature. Par exemple, les auteurs de [78, 92] proposent d'ajouter de nouveaux classifieurs pour traiter les nouvelles données, au risque de se retrouver avec un très grand nombre d'entre eux. Dans [93, 76], les auteurs s'appuient sur des machines à vecteurs de support qu'il est nécessaire de ré-entraîner lors de l'acquisition de nouvelles données, générant de l'oubli catastrophique [46, 17]. Afin de répondre à ces deux problèmes, une combinaison de machines à vecteurs de support avec l'algorithme *learn++* a été proposée [16, 68]. Cette combinaison offre des performances prometteuses [68]. Cependant, elle requiert l'entraînement systématique d'un classifieur s'appuyant sur les nouvelles et anciennes données, et certaines informations sont oubliées alors que de nouvelles sont apprises.

Récemment, dans [81] les auteurs ont proposé une méthode d'apprentissage incrémental appelée "Incremental Classifier and Representation Learning" (iCaRL), basée sur un extracteur de caractéristiques DNN entraînable, suivie d'une couche de classification. Dans [66], les auteurs ont proposé d'utiliser un DNN pré-entrainé auquel aucun changement n'est apporté durant la phase d'apprentissage, comme extracteur de caractéristiques suivi d'un Nearest Class Mean classifier (NCM). NCM représente chaque classe à l'aide du vecteur caractéristique moyen calculé à partir de tous les exemples observés jusqu'à présent et appartenant à cette classe. Le processus de classification se fait en attribuant la classe du vecteur moyen le plus semblable à l'aide d'une métrique qui peut être apprise à partir des données. Finalement, dans [27] et [7], nous avons introduit Budget Restricted Incremental Learning (BRIL) et Transfer Increment Learning with Data Augmentation (TILDA), deux méthodes incrémentales utilisant de l'apprentissage par transfert suivit d'un classifieur incrémental visant à réduire la complexité de la phase d'apprentissage tout en gardant une justesse acceptable. En appliquant la segmentation sur les vecteurs caractéristiques obtenus grâce à l'apprentissage par transfert, la justesse de NCM, BRIL ainsi que TILDA peut être améliorée faisant plus particulièrement de TILDA une solution incrémentale, atteignant une justesse comparable à des méthodes non-incrémentales et facilitant l'apprentissage sur des systèmes embarqués aux ressources limitées.

Conclusion et ouvertures

Dans ce manuscrit, nous avons abordé essentiellement le problème de la mise en œuvre de solutions d'apprentissage en profondeur dans le contexte des systèmes embarqués à ressources limitées. Nous avons examiné plusieurs propositions visant à réduire à la fois la mémoire et le nombre d'opérations, à l'aide de l'élagage, de la quantification ou de la factorisation. Nous avons vu comment réduire la consommation d'énergie d'un système embarqué en réduire la tension d'alimentation tout en gardant une justesse acceptable.

Cependant, de telle méthodes sont uniquement adaptées à la phase d'inférence et ne peuvent réduire la complexité ou la mémoire nécessaire durant la phase d'entraînement.

Nous avons également introduit de nouvelles méthodes d'apprentissage incrémental, puisqu'un modèle d'apprentissage profond basique n'a pas la capacité d'apprendre de nouvelles informations au fur et à mesure sans détruire les connaissances acquises ou apprises précédemment. De telle méthodes peuvent être considérées comme des solutions alternatives visant à faciliter la phase d'entraînement ou d'apprentissage, donnant ainsi des solutions d'apprentissage incrémental sur puce.

Nos travaux ainsi que d'autres méthodes de l'état de l'art qui visent à remplacer la convolution par un décalage de l'entrée suivie d'une multiplication ouvrent une nouvelle perspective considérable. Les réseaux de neurones convolutifs ont été considérés comme la meilleure solution applicable aux ensembles de données de traitement contenant des images. Cependant, dans ce manuscrit, nous avons montré que les méthodes basées sur les couches à décalage peuvent être plus performantes que les CNN dans certaines conditions.

Les méthodes de quantification présentées dans ce manuscrit visent à réduire la mémoire et le nombre d'opérations uniquement durant la phase de classification (ou d'inférence). La phase d'apprentissage étant plus coûteuse, reconsidérer ces méthodes et leur utilisation pour réduire la mémoire et le nombre d'opérations pendant l'entraînement serait une contribution importante dans ce domaine. Cette question devrait certainement susciter plus d'intérêt, car il est tout à fait clair que de nombreuses applications de l'apprentissage profond nécessiteront un réglage fin des paramètres à la volée.

Enfin, nous pensons que l'apprentissage sur puce sera un des prochains sujets majeurs du domaine. En particulier, la recherche d'une solution pour l'entraînement d'algorithmes d'apprentissage profond sur un système embarqué avec des ressources limitées comme les smartphones ou les FPGAs semble cruciale à court terme. En effet, une telle solution vise à remplacer les GPUs ou les TPUs, des dispositifs chers et coûteux en terme d'énergie, par des systèmes embarqués pour entraîner les réseaux de neurones. Une telle solution pourrait exploiter nos contributions sur les architectures matérielles et les méthodes de quantification pour réduire la mémoire et le nombre de d'opérations de la phase d'inférence comme point de départ, afin de les réadapter pour proposer une solution d'apprentissage sur puce. Ainsi, l'apprentissage sur puce fournirait une solution moins coûteuse pour entraîner des réseaux de neurones sur des appareils moins chers (smartphones ou FPGA) accessibles à tous, à faible consommation énergétique.

Chapter 1

Introduction

Machine learning refers to the field of computer science in which the principle is to learn from examples, experiments and/or interactions. Instead of being explicitly hard-coded to perform a specific task, algorithms developed in this field are thus able to acquire their functionality in a way that is without doubt much closer to the way humans learn. As such, machine learning is a subfield of artificial intelligence. There are many reasons to motivate machine learning. For example, in some cases there is no known explicit solution to the problem, like for image classification. In some other cases, the known solutions are too computationally expensive, and machine learning brings interesting trade-offs between correctness and speed of the algorithms.

Thanks to the interest machine learning received during the last two decades, it has become a very mature field and the state-of-the-art in numerous challenging domains such as computer vision or natural language processing, surpassing even human capacities for some tasks. For instance, in 2016, a machine learning based solution has been introduced with a better ability to classify and recognize objects than human [95]. Moreover, during the same year, another machine learning method called AlphaGo defeated world's champions in the GO game [88].

Machine learning is not a method by itself, but a set of different methods such as Support Vector Machine (SVM), Random Forest and deep learning. The latter is the one that received the most interest during these last years. It is built upon a brain-inspired algorithm called artificial neural network, in which neurons are connected and exchange information between them. Recent applications are more focused on using deep learning instead of other machine learning algorithms for several reasons. The first one is that deep learning is one of the few methods we know today that is able to exploit the statistical dependencies hidden in massive amounts of data, where other machine learning

methods can quickly reach a saturation point as depicted in Figure 1.1. This is arguably due to the fact that the complexity of training a deep learning architecture scales linearly with the number of elements in the dataset, making it the only viable option for very large datasets such as the ones defined in Chapter 2. In addition, deep learning based solutions have the ability to decompose a difficult problem in a composition of simpler ones, all trained simultaneously. As such, most of the deep learning methods directly handle raw data, while other methods require feature extraction defined by human experts (*cf.* Figure 1.2). It is well known that in the field of computer vision, the adoption of deep learning began with the understanding that in the classical decomposition of learning methods in two steps – feature extraction then classification –, little progress was to be expected on the last step.

Obviously, deep learning is not the ideal solution for every problem. Throughout this thesis, we shall deeply question its computational and memory costs, making it sometimes impractical for resource-limited devices or real-time processing applications. Also, because it relies on a very large number of parameters that are trained through optimization routines, the understanding, interpretation and robustness of deep learning raise a lot of concerns and questions for which it is fair to say they remain mostly open. In application domains such as automatically assisted surgery, or autonomous cars, these questions are a main barrier to the global adoption of the methodology.

As most machine learning methods, deep learning is usually made of two phases. The first one is the learning phase (also called training phase), where the learning parameters are tuned in order to solve a given task. The second one is the predicting phase (also called classification phase or inference), where the model is used to predict and classify the output corresponding to a given input for a given task. For instance, if during the learning phase the deep learning model learns to differentiate animals from cars, during prediction it will predict if a given previously unseen input corresponds to an animal or a car.

Due to its state-of-the-art performance, deep learning is now pervasive in many applications and domains, and has become a part of our daily life and tasks, even though we do not necessarily realize using it. Among the most impressive and challenging applications of deep learning, we find:

1. **Image recognition and detection**:

   Thanks to deep learning we can recognize and detect the position of objects, animals or even people into a picture or a video with a high accuracy (*cf.* Figure 1.3).

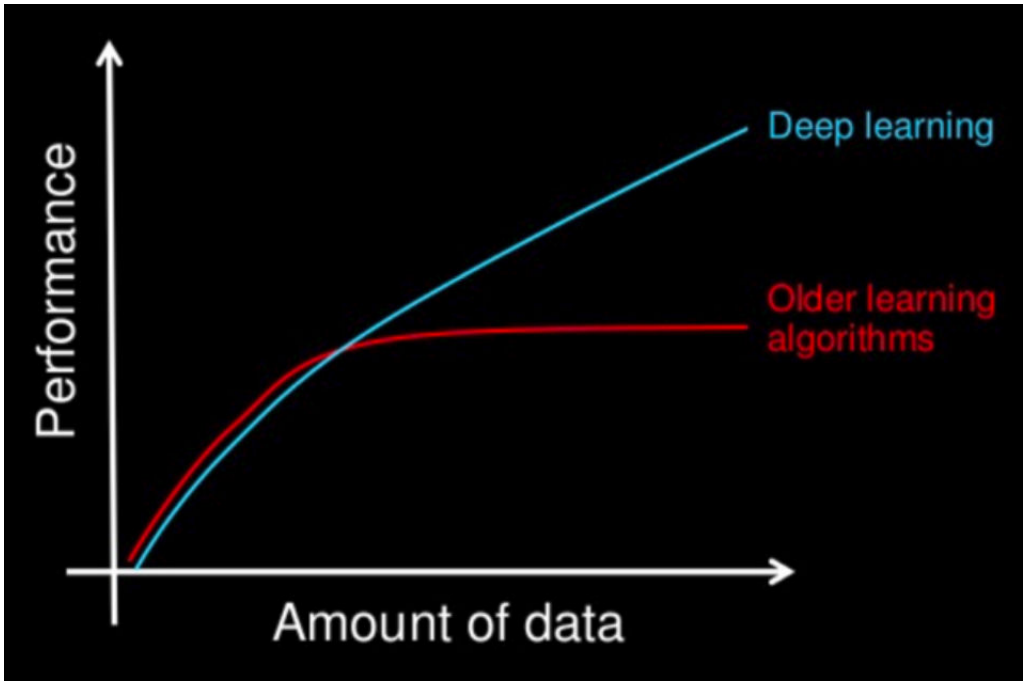


Figure 1.1: How machine learning techniques scale with amounts of data[a].

[a]https://www.slideshare.net/ExtractConf

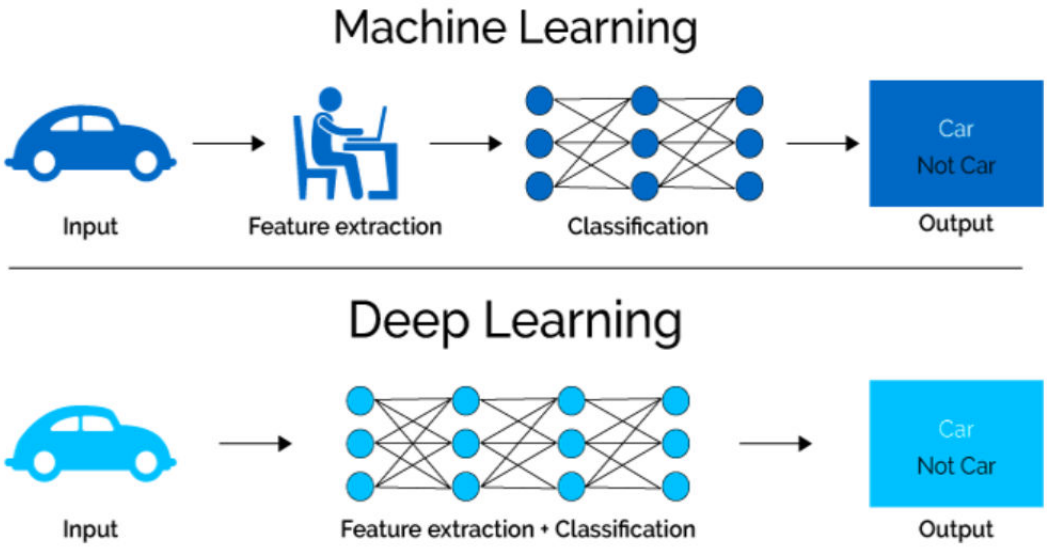


Figure 1.2: Feature extraction in deep learning and in general machine learning methods[a].

[a]https://medium.com/intro-to-artificial-intelligence/deep-learning-series-1-intro-to-deep-learning-abb1780ee20

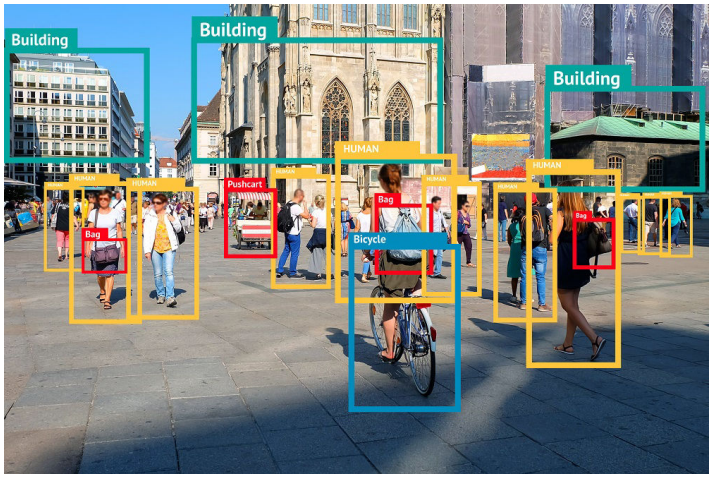


Figure 1.3: Objects recognition and detection using a deep learning solution[a].

[a]https://www.technative.io/all-seeing-ai-video-analytics-in-action

2. **Natural language processing**:

   Deep learning is used in natural language processing to extract the meaning of a given word, or a sentence, and analyze it. For instance, it is possible to analyze what is said in a Google review to determine whether it is positive or negative (a.k.a. sentiment analysis).

3. **Playing games (AlphaGo)**:

   Go is a strategic and complex Chinese board game and was one of few games where human were still better than machines until 2014. Developed by the British company Deep Mind, AlphaGo, a deep learning based algorithm, defeated in 2015 the Go world champion.

4. **Positive hopes**:

   Deep learning is used in the medical domain, since it can be combined to medical imaging to improve cancer diagnosis by extracting some important details into images that cannot be detected by the human eye[1]. On another hand, deep learning can also be used to help fight climate change[2]. Indeed, there are years of climate-related and weather data available that can be used by deep learning for better decision making. For instance, deep learning gives a more accurate weather prediction than humans, can detect earlier warning signs of a catastrophic weather event and thus reduces damage to human lives[3]. Deep learning is also used in

[1]https://experiences.microsoft.fr/business/intelligence-artificielle-ia-business/intelligence-artificielle-medecine

[2]https://bernardmarr.com/default.asp?contentID=1360

[3]https://www.globaltechcouncil.org/artificial-intelligence/how-can-deep-learning-solve-the-problem-

education, to detect students strengths and weaknesses and adapt and review students learning path[4]. For instance, the mobile application Duolingo uses a deep learning based solution to predict the probability of remembering particular words, and then offers to more practice words which are harder to remember[5].

To achieve sate-of-the-art performance, deep learning uses a large amount of resources, including memory to store models and data, and computations to process inputs, leading to a large energy consumption. Such needs can quickly become a limitation that reduce deep learning application domains. Memory, computation and power represent key resources that recently introduced deep learning methods aim to preserve. There are scientific, technical and even societal challenges associated with these questions.

1. **Societal challenges**:

   In societal challenges, two main subjects can be discussed, the relation between ecology and deep learning, and accessibility of deep learning to everyone. As mentioned above, deep learning needs a large memory footprint and computations to store and process data, especially during learning where the algorithm needs to repeat the process several times trying to find the structure connecting the artificial neurons between them that allows to reach the best performance. Considering that, almost all deep learning applications and research use Graphics Processing Units (GPUs), a significant energy consumption device, during hours, days or sometimes months. The energy cost can quickly become huge. Such an energy consumption makes deep learning an expensive solution which does not respect the environment and sustainable development.

   It is very difficult to obtain objective indicators about the energy consumption that is dedicated in datacenters to the computations using deep learning methods. But it is fair to envision that the usage is growing, and that it is definitely not insignificant. At the time of writing this thesis, training a modern deep learning architectures on the celebrated ImageNet ILSVRC 2012 challenge requires of the order of one week of computations on a modern desktop computer. As this benchmark is often required to prove the efficiency of methods when submitting a paper to a major and well known conference, a lot of hyperparameters have to be tried, hence as many weeks or even months of computations. Knowing that the power consumption of such a computer is of the order of 1000W, one can quickly derive

of-global-climate-change/

[4] https://aibusiness.com/machine-learning-and-the-future-of-education

[5] https://www.forbes.com/sites/bernardmarr/2018/07/25/how-is-ai-used-in-education-real-world-examples-of-today-and-a-peek-into-the-future/70626870586e

that most papers submitted to top-tier conferences used computations corresponding to more than one year household consumption in a typical occidental country. Of course the point of this discussion is not to criticize research or the way it is conducted right now, but simply to illustrate how ecologically impactful simple computations can become. When it comes to big companies, one has to imagine orders of magnitudes more demanding architectures.

Finding methods to reduce the power consumption of trained architectures, as well as the training cost, could be key to limiting the ecological impact the field has and is going to have in the coming years.

Also, deep learning solutions aim at assisting people in their work or daily life, and thus relieve them from some exhaustive work and ease their daily tasks. However, and as mentioned above, deep learning is an expensive solution which requires a large memory footprint, computations and power usage, and uses GPUs, an expensive device to process data. Such needs make the accessibility of deep learning to everyone a considerable challenge, and then may not reach its objective which is assisting people in their work and daily life. Indeed, if data is a key limiting factor for public research institutions, computations also are. By reducing the resources needed to find the correct hyperparameters for a given task, we would make a step forward more democratization of deep learning for everyone.

2. **Technical challenges**:

   Technical challenges may occur when using deep learning solutions in real time applications or implementing them on limited resources embedded systems. Indeed, to process a given input, the algorithm needs to read deep learning model's parameters from a memory, and computes some basic operations using these parameters and the input. Due to the large memory needed to store deep learning model and computations needed to process data, the algorithm needs to read model's parameters from the memory numerous times, to compute a large number of operations and to store the result of each operation in the memory. Therefore, such an algorithm requires a significant amount of time to process data. To achieve a state-of-the-art performance, deep learning models rely on a large number of parameters and computations which increases the time needed to process a given input. Thus, using deep learning methods for real time applications can be challenging.

   Another technical challenge when considering real time applications would be incremental learning (also called continuous learning or curriculum learning), a learning scenario in which new pieces of information are learned through time, building over previously acquired knowledge. Despite the fact that deep learning models are brain inspired, they are not adapted to incremental learning, since when learning

new information, models are adapted to better represent the new learned data, and then previously learned knowledge is destroyed. Note that this phenomenon is referred to as "catastrophic forgetting" in the literature [46, 17]. Thus, deep learning may not be adapted to a real time application during which data streaming continuously provides previously unseen information.

Embedded systems with limited resources such as smartphones or more low level ones such as Field-Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuit (ASIC) need to address some technical challenges in order to use deep learning solutions. Indeed, embedded systems have limited computational resources and scarce amounts of memory. As a consequence, embedded systems are not adapted to store large parameters sets required by modern deep learning models, and cannot perform the extensive computations required by the model in a reasonable time. Finally, such embedded systems are battery-powered, which further limits the feasibility of implementing algorithms with intensive memory access and computations. For all these reasons, implementing state of the art deep learning applications on embedded systems is currently challenging.

3. **Scientific challenges**:

   Deep learning is mostly an experimental field, where results and improvements are reached thanks to experimental protocols. Therefore, finding the deep learning architecture that achieves the best performance, can be a demanding search where all possible structures need to be tested.

   A scientific challenge would be to describe deep learning models using some mathematical assumptions. Indeed, such assumptions allow to understand deep learning models, and then accelerate model's structure search, since they assert which structure is more relevant to achieve the best performance for a given task. A mathematical assumption can be used to define the perfect number of artificial neurons in a deep learning model, the way they are initially connected (before learning), the number of iterations the model needs to process and learn the same data, and the algorithm used during learning to refine neurons connections. Thus, it avoids to test all possible cases for each parameter, which drastically accelerates and eases the model structure search.

   Usually, an artificial neural network (or deep learning model) contains a large number of neurons and connections, which makes it a complex structure, difficult to understand or to mathematically describe. A relevant approach to ease understanding deep learning models is to rely on models containing fewer parameters and computations. However, obtaining a comparable state-of-the-art performance using a less complex model is a real challenge. Moreover, it is a necessary criterion,

otherwise a not suitable structure with lower performance will be studied, giving no information about the suitable model.

In this thesis, we focus on reducing memory and computations of deep learning since they are the two main limitations that beget the different challenges, and tackle the problem of predicting and learning on chip. We review and introduce some methods that aim at reducing deep learning model size and computations, and others able to perform incremental learning, in order to address all the challenges discussed above.

The outline of this Ph.D. thesis is as follows:

- First, in Chapter 2 we introduce all the notions required to describe our works and other related ones. In more details, we first introduce the different used datasets, then we define basic functions used to build neural network structures, and finally we explain the learning process.
- Then, in Chapter 3 we focus on quantizing neural networks and reducing their size. More precisely, we first review state-of-the-art methods that aim at quantizing and reducing neural networks size, then we introduce our contribution and compare it to other methods. Next, we present a hardware architecture to implement our contribution on an FPGA, and finally we study the effect of reducing the energy consumption of a device on deep learning performance.
- Next, in Chapter 4 we discuss incremental learning. Actually, we review state-of-the-art incremental learning methods, then we present and compare our contribution with other methods. Finally, we propose a hardware architecture to implement our method on FPGA to obtain an incremental learning on chip solution.
- Finally, in Chapter 5 we summarize the different contributions of this thesis, conclude and discuss future work.

In this manuscript, we use a Xilinx Ultra Scale Vu13p (xcvu13p-figd2104-1-e) Field Programmable Gate Array (FPGA) as a reference to evaluate hardware implementations. It is worth to mention that such a choice is made since this is one of the most recent and largest FPGAs available in our lab, able to compete with latest CPUs and GPUs.

The scientific contributions that were written during this PhD are:

- Hacene, G. B., Gripon, V., Farrugia, N., Arzel, M., Jezequel, M. (2017, February). Finding All Matches in a Database using Binary Neural Networks. In COGNITIVE

2017: The Ninth International Conference on Advanced Cognitive Technologies and Applications (pp. 59-64).

- Medjkouh, S., Xue, B., Hacene, G. B. (2017, February). Sparse Clustered Neural Networks for the Assignment Problem. In COGNITIVE 2017: The Ninth International Conference on Advanced Cognitive Technologies and Applications (pp. 69-75).

- Hacene, G. B., Gripon, V., Farrugia, N., Arzel, M., Jezequel, M. Budget Restricted Incremental Learning with Pre-Trained Convolutional Neural Networks and Binary Associative Memories. In SIPS 2017: International Workshop on Signal Processing Systems.

- Hacene, G. B., Gripon, V., Farrugia, N., Arzel, M., Jezequel, M. Incremental Learning on Chip. In GlobalSIP 2017: Global Conference on Signal and Information Processing.

- Marques, M. R. S., Hacene, G. B., Lassance, C. E. R. K., Horrein, P. H. (2017, July). Large-Scale Memory of Sequences Using Binary Sparse Neural Networks on GPU. In High Performance Computing Simulation (HPCS), 2017 International Conference on (pp. 553-559). IEEE.

- Gripon, V., Hacene, G. B., Lowe, M., Vermet, F. (2018, April). Improving Accuracy of Nonparametric Transfer Learning Via Vector Segmentation. In 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (PP. 2966-2970). IEEE

- Hacene, G. B., Gripon, V., Arzel, M., Farrugia, N., Bengio, Y. (2018). Quantized guided pruning for efficient hardware implementations of convolutional neural networks. arXiv preprint arXiv:1812.11337.

- Hacene, G. B., Lassance, C., Gripon, V., Bengio, Y. (2019). Attention Based Pruning for Shift Networks.

- Bontonou, M., Lassance, C., Boukli Hacene,G., Gripon, V. INTRODUCING GRAPH SMOOTHNESS LOSS FOR TRAINING DEEP LEARNING ARCHITECTURES. In 2019 IEEE Data Science Workshop (DSW).

- Boukli Hacene, G., Gripon, V., Farrugia, N., Arzel, M., Jezequel, M. (2018). Transfer Incremental Learning Using Data Augmentation. Applied Sciences, 8(12), 2512.

- Hacene, G. B., Leduc-Primeau, F., Soussia, A. B., Gripon, V., Gagnon, F. Training Modern Deep Neural Networks for Memory-Fault Robustness. In 2019 IEEE International Symposium on Circuits and Systems (ISCAS).

- Boukli Hacene, G., Gripon, V., Farrugia, N., Arzel, M., Jezequel, M. (2019). Efficient Hardware Implementation of Incremental Learning and Inference on Chip. In 2019 IEEE International NEWCAS Conference.

Chapter 2

Basics in Deep Learning

In this chapter, we introduce some notions and definitions related to our domains of interest. We first introduce deep neural networks (DNNs). We then explain how to apply them to challenging computer vision datasets introduced in Section 2.1.

Since a DNN architecture can be complex and contains numerous layers and functions, we first define in Section 2.2 the basic DNN components, using formalism of tensor spaces. Next, we introduce in Section 2.3 how to assemble such components to obtain neural networks, and some classical DNN architectures. We finally present the learning and inference processes and discuss performance on the abovementioned datasets. Note that here we only provide a general overview of the field, while focusing in particular on the concepts that will be further developed in the next chapters. The reader can refer to textbooks such as [4] for generalities in machine learning, as well as [20] for a more in-depth presentation of deep learning.

2.1 Datasets

We present in this Section the datasets used to perform experiments in Chapters 3 and 4.

2.1.1 Training, Validation and Test Sets

To assess the performance of a classifier, it is common to rely on a methodology that consists in using two datasets made of pairs of the form `(input image, corresponding label)`. The first one, called training set is used to train the classifier. The second one, called validation set, is used to assess the ability of the trained classifier to generalize

to novel unseen inputs. Even though this is not the main motivation of this document, it is worth mentioning that this methodology, often referred to as "crossvalidation" in the literature, is more and more criticized in the community. As a matter of fact, it has been known for years that trained architectures that appear to achieve very good performance in generalization, as assessed using the validation set, can be very easily fooled using imperceptible changes of their inputs [64]. This can be easily explained by the fact deep learning architectures, which are the state-of-the-art classifiers on these datasets, are made of a huge number of parameters that are likely to capture biases of the training set. These biases are likely to also exist in the validation set, since in most cases both are sampled from the same distribution.

There is a third type of dataset called test set. Usually the training set is used to train the classifier, the validation set is used to test the classifier's generalisation (*ie.* if the classifier performs well on other unseen data), and the test set is an unlabelled and unknown dataset classified and labelled by the classifier. Note that in some cases, the validation set and test sets are the same. The generalisation can be defined as the ability of a classifier to avoid over-fitting [115] when considering the same data distribution into validation and test sets as into training set. On another hand, the generalisation can be defined as the robustness of a classifier against adversarial examples [112] when validation and test set distributions are different from training set one (*eg.* using a low coast camera during classification phase that provides a low quality images comparing to high quality training images, specially when considering mobile applications ruining on embedded systems). In such a scenario, the generalisation is more challenging since the classifier is not well adapted to this new and unseen data distribution. To measure how good the generalisation is, a measurement called accuracy is used, and which represents the ratio of number of correct predictions to the total number of input samples [4]. Note that in some cases, the accuracy reported is refereed to by top-k accuracy, which means that if the expected answer matches one of the classifier's k highest probability answers, then it is considered as a correct prediction.

2.1.2 CIFAR10 and CIFAR100

CIFAR10 and CIFAR100 are datasets containing colored tiny pictures of size 32×32 [50]. Because they are encoded using the three main colors, a picture in one of these datasets can be represented as a tridimensional tensor containing a total of $32 \times 32 \times 3 = 3072$ dimensions. CIFAR10 contains 10 classes, each one made of 5000 images for training and 1000 images for testing. CIFAR100 contains 100 classes, each one made of 500 images for training and 100 images for testing. These datasets are widely accepted

as an interesting compromise between a toy dataset, in the sense that the images are small, and as such training architectures can be fast, and a competitive one, as the best performance reported in the state-of-the-art is respectively of 97.6% accuracy for CIFAR10 [122] and only $85,42\%$ for CIFAR100 [63].

2.1.3 ImageNet (ILSVRC 2012)

ImageNet is a large visual dataset used in visual object recognition research. It is made of more than 14 millions of images and $20,000$ classes. ILSVRC2012 [85] is a subset of Imagenet that contains $1,000$ classes, more than $1,200,000$ images for training and $50,000$ images for testing. Contrary to CIFAR10 and CIFAR100, the images have various sizes which are typically of the order of a 1,000 pixels in both width and height. It is common to resize the input images to 200 to 300 pixels square inputs that are being processed by the classifier. Despite being a few years old, ILSVRC remains a highly competitive benchmark that requires a processing time of the order of days to weeks to be trained. As such, it is considered by most as a reference in vision benchmarks.

2.1.4 ImageNet1, ImageNet2 and ImageNet50

In this document we introduce two other datasets extracted from Imagenet. We call them ImageNet1 and ImageNet2. Both contain 10 classes, distinct between themselves and from that in the ILSVRC dataset. Each class contains about 900 images for training and 100 for testing. In some cases, we also make use of ImageNet50, built using the same idea, but containing a total of 50 classes.

2.1.5 AudioSet

AudioSet is a large dataset made of 10 second sound clips extracted from YouTube videos [18]. It contains more than 2 millions of samples which correspond to 5.8 thousands of hours of audio split into 527 classes. AudioSet is sometimes presented as the equivalent of ImageNet for sound recognition.

Let us point out that these datasets are but a small fraction of the plethora that can be found freely online. In order to be fair in comparisons, it is crucial that different methodologies are evaluated against using the same benchmarks. This is why all the results presented in this manuscript use these few selected datasets.

2.2 Main Elements

Deep Neural Networks are complex mathematical objects that are built by assembling simpler elementary blocks. This is why we first introduce these basic blocks. Namely, in this section we introduce some activation functions, loss functions and common layers.

2.2.1 Activation Functions

An activation function f is a non-linear and differentiable function usually applied to a layer output. Its main role is to introduce non-linearity between layers, and thus to avoid factorizing the whole network into a single linear operation. Indeed, recall that the algebra of tensors is associative.

Common activation functions used in a neural network include:

- Relu or ReLU (Rectified Linear Unit): the input x is a scalar, and the output x' is computed as follows:
$$f(x) = x' = \max(0, x).$$
- Sigmoid: the input x is a scalar, and the output x' is computed as follows:
$$f(x) = x' = \frac{1}{1 + e^{-x}}.$$
- tanh: the input x is a scalar, and the output x' is computed as follows:
$$f(x) = x' = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$
- Softmax: the input $\mathbf{X} = \{x_1, x_2, \ldots, x_D\}$ is a vector with dimension D, and the output $\mathbf{X}' = \{x'_1, x'_2, \ldots, x'_D\}$ is a vector with same dimension computed as follows:
$$f(\mathbf{X})_i = x'_i = \frac{e^{\frac{x_i}{T}}}{\sum_{j=1}^{D} e^{\frac{x_j}{T}}}.$$
where T is called the softmax temperature. Note that when the temperature tends to 0, the softmax tends to a hard maximum indicator.

2.2.2 Loss Functions

Let us consider the DNN's output $\mathbf{X}_L = \{x_{L,1}, x_{L,2}, \ldots, x_{L,Y}\}$ associated with the input $\mathbf{X}_0 = \{x_{0,1}, x_{0,2}, \ldots, x_{0,D}\}$ through a given DNN. Here, D refers to the dimension of the

input, Y to that of the output (typically Y is the number of classes in the problem), and L to the number of layers in the architecture. A loss function g (also referred to as cost function) evaluates how far this output is from an expected target $\mathbf{Y} = \{y_1, y_2, \ldots, y_{,Y}\}$. In other words, it measures an error when predicting the class of a given input.

Common loss functions used to train neural networks are:

- Mean Square Error:
$$g(\mathbf{X}_L, \mathbf{Y}) = \frac{1}{Y}\sum_{i=1}^{Y}(x_{L,i} - y_i)^2.$$
- Cross Entropy:
$$g(\mathbf{X}_L, \mathbf{Y}) = -\frac{1}{Y}\sum_{i=1}^{Y} y_i \log(x_{L,i}).$$
- Binary Cross Entropy:
$$g(\mathbf{X}_L, \mathbf{Y}) = -\frac{1}{Y}\sum_{i=1}^{Y} y_i \log(x_{L,i}) + (1 - y_i)\log(1 - x_{L,i}).$$
- Hinge loss:
$$g(\mathbf{X}_L, \mathbf{Y}) = \frac{1}{Y}\sum_{i=1}^{Y} \max(0, 1 - y_i x_{L,i}).$$

Mean Square Error (MSE) was originally the first of these losses to be introduced. It very intuitively measures the L_2 distance between the output of the DNN and the expected target. A key problem with using MSE is that it tends to slow the training procedure when the error becomes small. However, cross entropy, binary cross entropy and hinge loss have the advantage of accelerating the convergence, in particular when the error becomes small. This is due to the properties of the gradients of these losses, as for instance cross entropy can only be used in conjunction with a normalization factor on the output, such as using the softmax activation.

2.2.3 Layers

The layer indexed by l is a combination of one (or more) linear function(s) h and one non-linear (or activation) function f. It computes an output $\mathbf{X}_{l+1}$ using an input $\mathbf{X}_l$, its learnable weights $\mathbf{W}_l$ and biases $\mathbf{B}_l$ as follows:

$$\mathbf{X}_{l+1} = f(h(\mathbf{X}_l, \mathbf{W}_l) + \mathbf{B}_l).$$

The layer type is defined by its linear function h. Note that l represents the index of the layer in the neural network, where $1 \leq l \leq L$, and L is the total number of layers. Note that for readability reasons, we disregard both the bias parameters $\mathbf{B}_l$ and activation functions f in the following definitions.

The most common layers used in the literature are:

Fully Connected layers

Given an input vector $\mathbf{X}_l \in \mathbb{R}^{C_l}$ and using the learnable weight parameters $\mathbf{W}_l \in \mathbb{R}^{C_l \times C_{l+1}}$, the fully connected layer (FC) computes the output $\mathbf{X}_{l+1} \in \mathbb{R}^{C_{l+1}}$ as follows:

$$x_{l+1,c'} = \sum_{c=1}^{C_l} x_{l,i} w_{l,c,c'}, 1 \leq c' \leq C_{l+1}.$$

Convolutional layers

In 2D convolutional layers, an input tensor $\mathbf{X}_l$ is typically tridimensional: $\mathbf{X}_l \in \mathbb{R}^{C_l \times H_l \times R_l}$. Here, C_l represents the number of input channels (also called feature maps), and H and R represent respectively the length and the width of a feature map $\mathbf{X}_{l,c,H_l,R_l}$ where $1 \leq c \leq C_l$. The weight parameters $\mathbf{W}_l \in \mathbb{R}^{C_{l+1} \times C_l \times S1_l \times S2_l}$ are referred to as filters, where C_{l+1} represents the number of output channels, and $S1_l \times S2_l$ represents the size of a kernel $\mathbf{W}_{l,c',c,S1_l,S2_l}$, where $1 \leq c \leq C_l$ and $1 \leq c' \leq C_{l+1}$. The convolutional layer computes output feature maps $\mathbf{X}_{l+1} \in \mathbb{R}^{C_{l+1} \times H_{l+1} \times R_{l+1}}$ as follows:

$$x_{l+1,c',h',r'} = \sum_{c=1}^{C_l} \sum_{s1=1}^{S1} \sum_{s2=1}^{S2} x_{l,c,s1+h',s2+r'} w_{l,c',c,s1,s2}.$$

Note that unless otherwise mentioned, in this manuscript convolution refers to 2-dimensional (2D) convolution.

Depthwise Separable Convolution layers

Depthwise Separable Convolution is a depthwise convolution followed by a pointwise convolution. In a depthwise operation, the convolution is applied on one channel at a time. Given an input tensor $\mathbf{X}_l \in \mathbb{R}^{C_l \times H_l \times R_l}$, depthwise convolution uses the filter $\mathbf{W}_l \in \mathbb{R}^{C_l, S1_l \times S2_l}$ to compute an output tensor $\mathbf{XP}_l \in \mathbb{R}^{C_l \times HP_l \times RP_l}$ as follows:

$$xp_{l,c,h',r'} = \sum_{s1=1}^{S1} \sum_{s2=1}^{S2} x_{l,c,s1+h',s2+r'} w_{l,c,s1,s2}.$$

The pointwise convolution is a standard convolution (as defined below) when kernel size $S1 \times S2 = 1 \times 1$. Thus, the pointwise convolution takes $\mathbf{XP}_l$ as input and uses the filter $\mathbf{WP}_l \in \mathbb{R}^{C_{l+1} \times C_l \times 1 \times 1}$ to compute the output $\mathbf{X}_{l+1} \in \mathbb{R}^{C_{l+1} \times H_{l+1} \times R_{l+1}}$.

Batch Normalization layers

Note that because we did not need it before, we disregarded batches in the previous definitions. But typically, multiple inputs are processed in parallel in the architecture, adding a dimension to all input and output tensors. Given a batch of M input tensors $\{\mathbf{X}_l^1, \mathbf{X}_l^2, \ldots, \mathbf{X}_l^M\}$ where $\mathbf{X}_l^m \in \mathbb{R}^{C_l \times H_l \times R_l}$ and $1 \leq m \leq M$, a batch normalization [41] layer (BN) normalizes the input layer (the batch) by adjusting and scaling the input tensors $\mathbf{X}_l^m$, and then computes output tensors $\mathbf{X}_{l+1}^m$, as follows:

$$\mu_{l,c,h,r} = \frac{1}{M} \sum_{m=1}^{M} x_{l,c,h,r}^m$$

$$\sigma_{l,c,h,r}^2 = \frac{1}{M} \sum_{m=1}^{M} (x_{l,c,h,r}^m - \mu_{l,c,h,r})^2$$

$$\bar{x}_{l,c,h,r}^m = \frac{x_{l,c,h,r}^m - \mu_{l,c,h,r}}{\sqrt{\sigma^2 + \epsilon}}$$

$$x_{l+1,c,h,r}^m = \gamma_{l,c,h,r} \bar{x}_{l,c,h,r}^m + b_{l,c,h,r},$$

where ϵ is a small positive number used for numerical stability, and $\Gamma_l \in \mathbb{R}^{C_l \times H_l \times R_l}$ and $B_l \in \mathbb{R}^{C_l \times H_l \times R_l}$ are learnable parameters optimized during learning process.

BN layers have been introduced for various reasons [41]. For one, they allow the outputs of a given layer to be normalized, avoiding explosion effects that considerably harden the training of the architecture. Also, they introduce competition between inputs, which is empirically demonstrated to improve the accuracy.

Pooling

A pooling layer aims at downscaling a given input $\mathbf{X}_l$. Pooling layers can be used to avoid overfitting since they compute large scale features, and then consider more general and abstract representations of data. It is commonly thought that such a process helps optimizing deeper layers parameters, since the deeper a layer is, the more abstract data used to optimize the layer parameters are. But probably the most compelling argument to use downsampling is to reduce the number of operations required in deep layers, that still typically concentrate most of them.

2.3 Deep learning

Deep learning is a set of machine learning methods using Deep Neural Networks (DNNs) to model, learn and process data at a high level of abstraction. In this section we will introduce some DNNs and Convolutional Neural Networks (CNNs) architectures. We will also discuss the learning process of DNNs and how their parameters are modified and tuned to better complete a specific task.

2.3.1 Deep Neural Networks

Originally, neural networks were introduced as a cascade of layers chaining linear and non-linear functions [54, 55, 84, 33]. Recently, novel and more complex architectures have been proposed to further increase the accuracy while reducing the number of operations and parameters [94, 30, 39, 121].

In [13], the authors claim that a two-layer neural network can be used as a universal function approximator. However, to end up with such an approximator, the number of neurons in the first layer (or hidden layer) should tend to infinity. Usually, a DNN contains more than two layers with finite number of neurons. In this manuscript, we only consider some DNN architectures such as multi-layer perceptrons (MLPs) or CNNs, and omit other architectures such as Recurrent Neural Networks or Deep Belief Networks.

Multi Layer Perceptron

A multi layer perceptron (MLP) is a DNN made only of fully connected layers [36], and in which we usually refer to its internal layers as hidden layers and its last layer as output layer (*cf.* Figure 2.1). An MLP can achieve an accuracy of 99.2% on the toy dataset MNIST [98], which is comparable to the state-of-the-art methods. However, such a DNN architecture shows quickly some limitations when considering more challenging and complex datasets. For instance, an MLP achieves an accuracy of 72.7% on CIFAR10 at most [9], where other CNN based methods can easily reach and exceed 90% of accuracy. Thus, in recent DNN architectures, an MLP is used at the end of a CNN as a classifier and not as the DNN itself [51, 89].

Convolutional Neural Network

As a basic definition, a convolutional neural network (CNN) is a DNN made of convolutional layers. A CNN can also contain FC layers for classification purpose and pooling layers to downscale data. One of the earliest CNN that was introduced is LeNet-

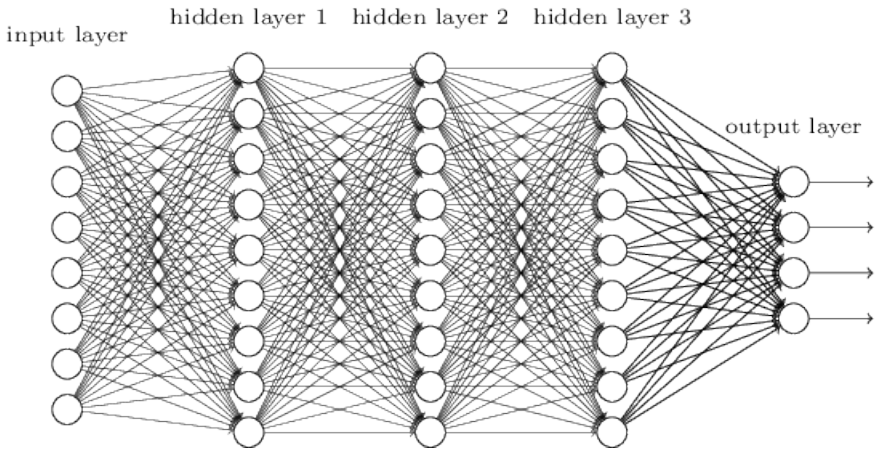


Figure 2.1: Multi Layer Perceptron (MLP)

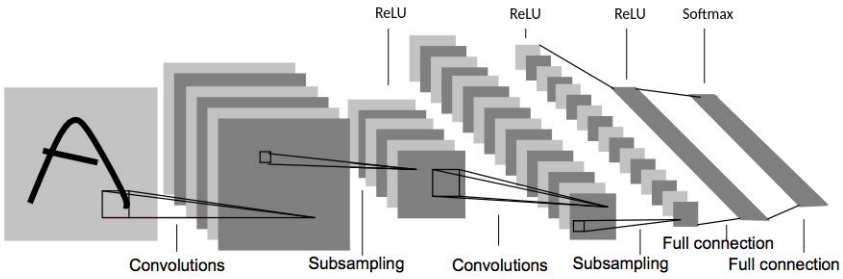


Figure 2.2: LeNet-5 architecture. Note that this figure is originally introduced in [55].

5 [55] which was used to classify handwritten digits and letters. LeNet-5 architecture is shown in Figure 2.2 and can be described as follows:

$$f_5 \circ h_7 \circ f_4 \circ h_6 \circ f_3 \circ h_5 \circ h_4 \circ f_2 \circ h_3 \circ h_2 \circ f_1 \circ h_1,$$

where h_1, h_3 and h_5 represent convolutional layers, h_2 and h_4 pooling operations, h_6 and h_7 fully connected layers, f_1 to f_4 Relu and f_5 a softmax activation. Based on LeNet-5 architecture, Krizhevsky *et al.* [51] propose Alexnet, a CNN architecture where layers are cascaded and which generates a surge of interest in the field since it represents the first CNN based solution that has won Imagenet competition (*cf.* Figure 2.3). In [89], the authors propose to improve Alexnet, and introduce VGG, another CNN architecture (*cf.* Figure 2.4). However, these CNN architectures show a limitation in accuracy even when adding more layers. To avoid such a drawback, recent works focus on different types of CNN architectures. In [30], the authors introduce Residual Networks (ResNet), based on a CNN architecture that uses residual connections, also referred to as skip connections, between different layers, so an upper layer can have one or more inputs coming from lower layers, and then providing more information to the upper layer (*cf.* Figure 2.5). ResNets containing hundreds of layers can be efficiently trained.

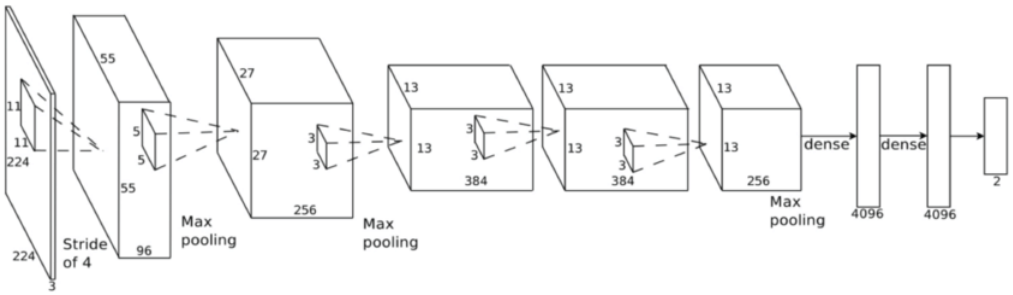


Figure 2.3: Alexnet architecture [51]. Note that "dense" in the figure refers to fully connected layer.

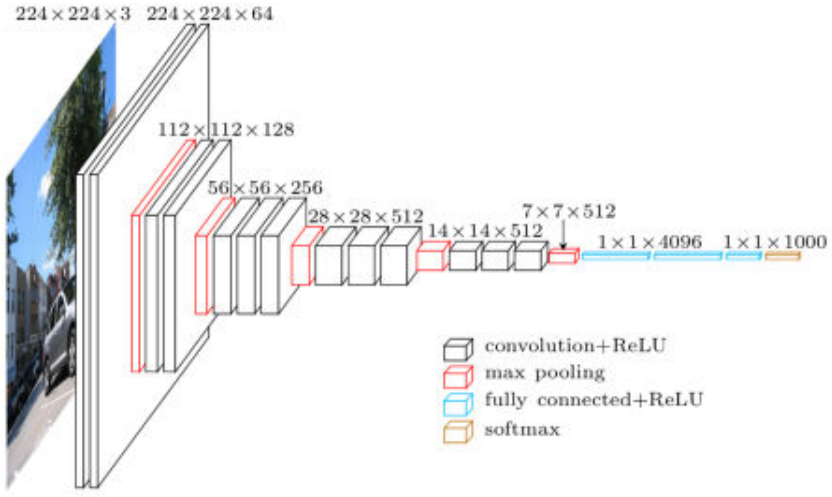


Figure 2.4: VGG architecture. Note that this figure is originally introduced in [89].

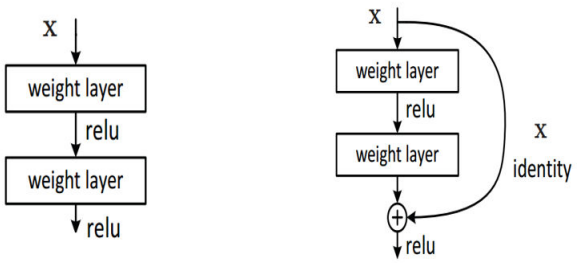


Figure 2.5: Comparison between a standard CNN component and a residual component.

Table 2.1: Comparison of accuracy between standard CNN architectures (Alexnet, VGG) and more recent and complex CNN architectures (ResNet, DenseNet, NASNet) on CIFAR10 and ImageNet ILSVRC2012.

Network	CIFAR10	ImageNet	
		Top-1	Top-5
AlexNet [51]	77.22%	56.6%	80.2%
VGG16 [89]	92.64%	71.93%	90.67%
ResNet-50 [30]	95.3%	79.26%	94.75%
DenseNet-121 [39]	95.04%	76.39%	93.34%
NASNet [63]	97.6%	82.7%	96.2%

Hung *et al.* [39] introduce Densely Connected Convolutional Networks (DenseNets), also based on a CNN architecture in which the input of an upper layer is the concatenation of all the outputs of lower layers (*cf.* Figure 2.8). Zoph *et al.* [122] propose to search for a block to build an efficient neural network architecture trained on a small dataset and then use this block to define a bigger DNN architecture trained on a larger dataset. Basically, the authors search for the best block (or cell) on CIFAR10, and then use the obtained cell on ImageNet dataset to define a more complex DNN containing more copies of this cell, each with its own parameters (*cf.* Figure 2.6 and Figure 2.7). Currently, NASNet architectures are considered as the state of the art in computer vision tasks such as ImageNet classification challenge [85], outperforming other CNN architectures, especially the simply chained layers based ones as depicted in Table 2.1.

It is worth mentioning that CNNs are getting more and more standard in vision benchmarks, whereas MLP being only used in other domains where no regular structure of signals is available. There are key properties of CNNs, that are going to be very important for the remaining of this document:

1. Convolutional layers can be applied to inputs with varying sizes. As such, it is possible to train CNNs using high resolution images and to deploy on smaller ones, or conversely. In other words, the number of parameters in convolutional layers is independent on both the input and output spatial dimensions of the images (but not of the number of input feature maps).

2. Most architectures introduced in the literature trade the spatial resolution for a higher number of feature maps, the deeper the layer is in the architecture. As such, layers close to the input typically contain a few number of feature maps, where

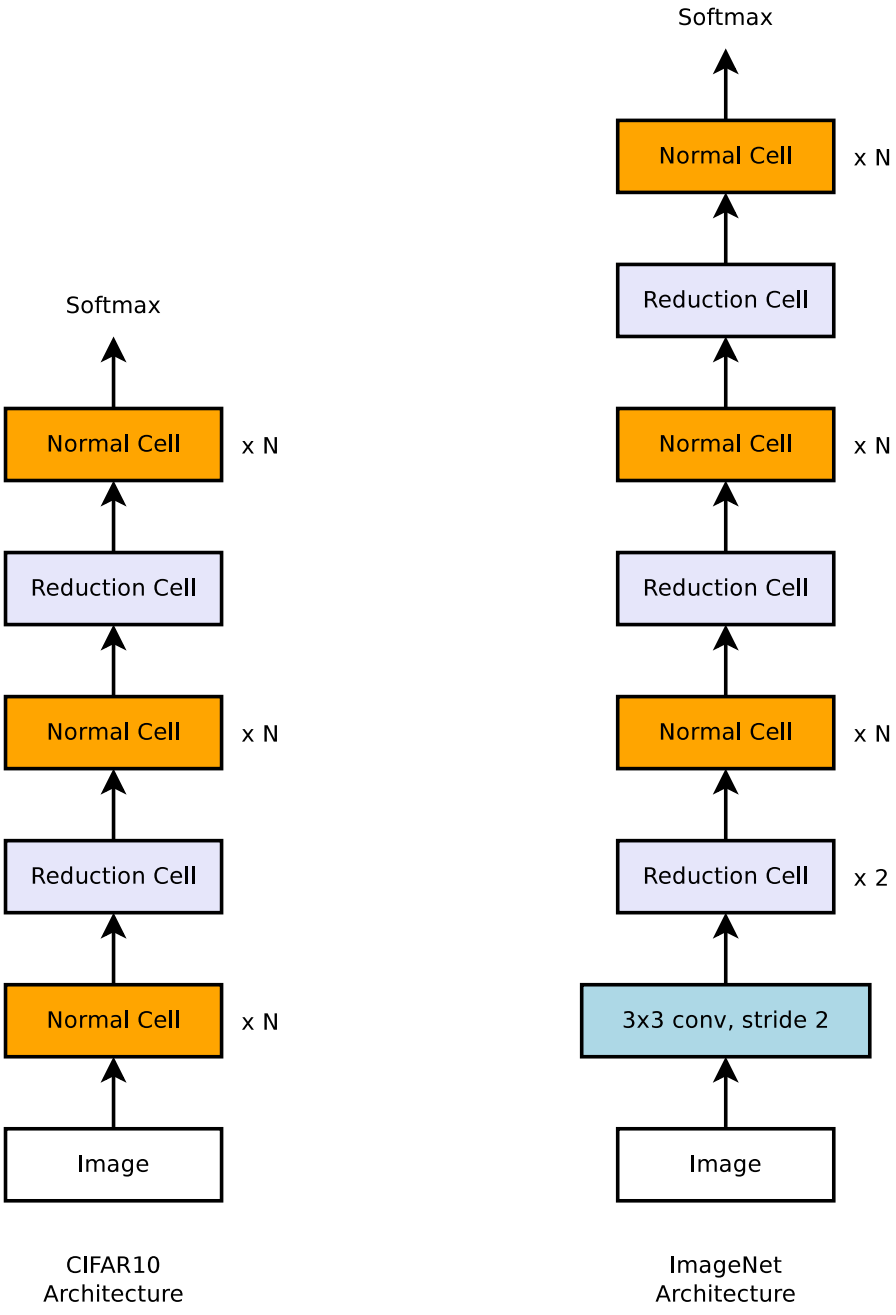


Figure 2.6: Overview of NASNet architecture where the obtained cells (normal cell and reduction cell depicted in Figure 2.7) on CIFAR10 are transfered to ImageNet. We notice that for ImageNet the authors use more reduction cells due to the size of images which is bigger than CIFAR10's. Note that this figure is originally introduced in [122].

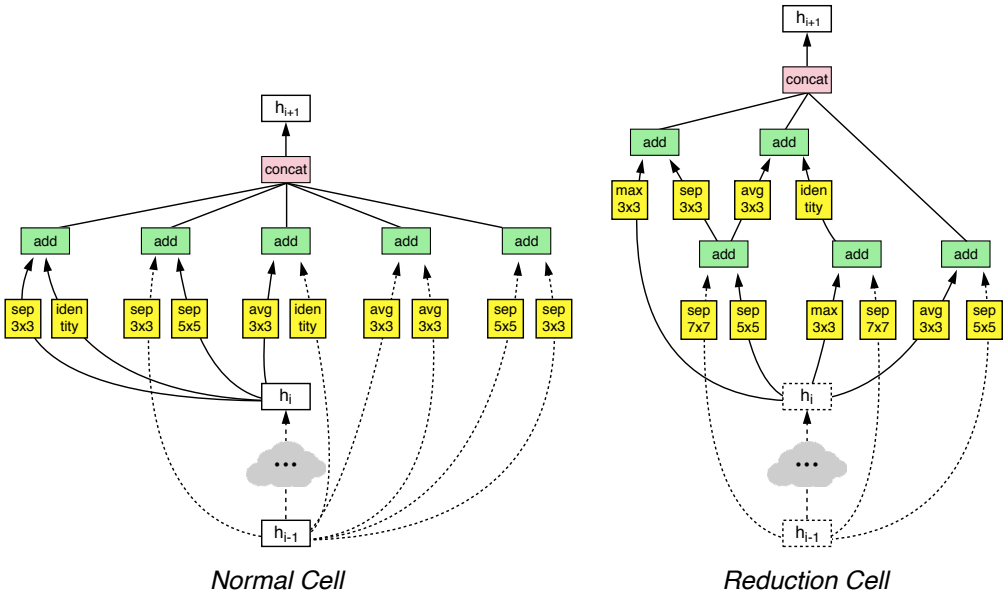


Figure 2.7: Overview of the normal cell architecture (left), and reduction cell architecture (right). Here sep refers to depth-wise separable convolution, max refers to max pooling and avg refers to average pooling. Note that this figure is originally introduced in [122].

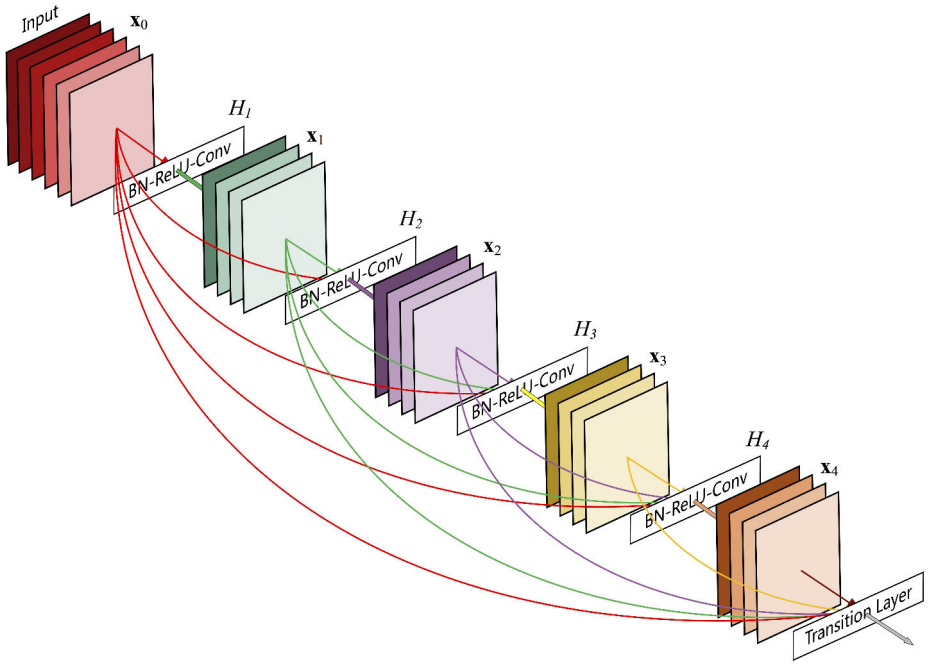


Figure 2.8: Overview of DenseNet architecture [39].

layers close to the output may contain thousands of those. This adjustment can be thought of as a way to avoid information bottlenecks.

3. The number of feature maps of each convolutional layer of a given DNN is considered as a hyperparameter. In many cases, authors scale this number proportionally for each layer, in order to adjust the accuracy *vs.* memory trade-off. These aspects will be closely looked at in the next chapters of this document.

4. Throughout numerous experiments, authors observed that it is often better to use more layers with smaller kernels for convolutions, rather than using larger kernels with few layers. The theoretical reasons for this finding are still highly unclear.

5. Convolutions are in most cases used jointly with data augmentation techniques, in which the training set is artificially increased by making small shifts, rotations and/or flips of input images.

2.3.2 Learning Process

The learning process objective is to minimize the loss of a given architecture on the training set. To do so, batches of inputs are processed, the loss function is computed on these inputs, and the result gradient error is back propagated throughout the whole architecture to update each weight concurrently [103, 84]. This process is typically split into two main parts: feed forward (or inference), where the output is computed for each input, and back propagation, where the weights are updated. These two steps are detailed in the following paragraphs.

Feed Forward

Given an input data $\mathbf{X}_0$ and its corresponding label $\mathbf{Y}$, the feed forward processes the input data through the L layers of the neural network, and obtains the neural network output $\mathbf{X}_L$. The loss function $g(\mathbf{X}_L, \mathbf{Y})$ is then used to evaluate the error made by the neural network relatively to the label $\mathbf{Y}$. Note that when considering a batch of M input data, the loss function computes the relative error as follows:

$$\bar{g} = \frac{1}{M} \sum_{m=1}^{M} g(\mathbf{X}_L^m, \mathbf{Y}^m).$$

Back Propagation

The learning process aims at modifying the DNN's parameters to reduce as much as possible the relative error computed by the loss function. To do so, gradient w.r.t w, denoted $\frac{\delta g}{\delta w}$ is used to update and optimize parameters using a gradient-descent based optimization algorithm at a learning rate α as follows:

$$w^{new} = w^{old} - \alpha \frac{\delta g}{\delta w^{old}}.$$

Usually, the gradients $\frac{\delta g}{\delta \mathbf{W}_l}$ are computed using the gradients w.r.t outputs of the next layer $l+1$ as follows:

$$\frac{\delta g}{\delta \mathbf{W}_l} = \frac{\delta \mathbf{X}_{l+1}}{\delta \mathbf{W}_l} \frac{\delta g}{\delta \mathbf{X}_{l+1}}.$$

On another hand, we have:

$$\mathbf{X}_{l+1} = f(h(\mathbf{X}_l, \mathbf{W}_l)) \Rightarrow \frac{\delta g}{\delta \mathbf{X}_l} = \frac{\delta f(h(\mathbf{X}_l, \mathbf{W}_l))}{\delta \mathbf{X}_l} \frac{\delta g}{\delta \mathbf{X}_{l+1}}.$$

This means the gradient calculation is back propagated from the last layer to the first layer of the neural network, in opposition of the feed forward process.

2.3.3 Classification Inherent Difficulties

Classification can be seen as a regression problem, in which the outputs are finite. Finding a solution that is able to generalize well is complex. And worse, in many cases, it is preferred a solution that contradicts some provided examples, if it yields more regularity.

In the more general context of machine learning, understanding what is a good generalization is an open challenge. As mentioned previously, most studies in the literature consider cross-validation a good proxy for assessing this generalization.

In this ambiguous context, a lot of techniques and methods introduced in the literature aim at improving generalization by constraining the structural properties of a DNN function, or by hardening the training process. Some examples include Dropout [90], where some output values are erased at random during the feed-forward step or L_2-regularization, where an additional term is added to the loss during training to penalize weights that diverge from 0.

In the literature, the overfitting refers to trained architectures that perform very well on the training set, but fail at generalizing to the test set. When using fully connected layers, this is often due to the fact they contain too many parameters, which allows the DNN to capture biases of the training set. It is important to point out that, due to the highly constrained nature of convolutional layers, increasing the number of parameters in convolutional layers typically does not create overfitting. This interesting property of convolutional layers is even more true when making use of data augmentation [2].

Chapter 3

Neural Networks and Low Resources Systems

Contents

3.1 Context

As we have seen in chapter 2, during the last few years Deep Neural Networks (DNNs) have made considerable progress and became state-of-the-art in various domains such

as natural language processing, sound/music classification, or computer vision. In particular, Convolutional Neural Network (CNN) architectures have continuously been improved to tackle new challenges such as image classification, object detection or face recognition, even to the point they are considered on par with human performance for some of these problems. However, such performance comes with a high cost in terms of the number of trainable parameters (memory) and the number of operations (computational complexity). As a consequence, the implementation of CNNs on embedded systems with limited resources is a difficult task.

In order to ease implementation of CNNs on resource-limited devices, authors have proposed several ways to reduce memory usage and/or number of operations. The main approaches are as follows. Some authors aim at using high level approaches and propose to use pruning techniques to reduce the number of connections in the architectures [56, 62, 32, 107] as described in Section 3.3, or factorisation techniques to merge several parts of DNN architectures [28, 105] as shown in Section 3.6. Other approaches use lightweight neural network architectures [40], grouped convolutions [37, 86], or decompose convolutional operations into shift operations followed by a point wise convolution [104, 44, 23, 26] as shown in Sections 3.4 and 3.5. In other works, authors propose to use low level approaches such as quantizing weight and/or activation values using n ($n < 32$) bits [101, 67, 118], up to the extreme cases where they become ternary [57] (usually -1,0, +1) or even binary (usually -1 and $+1$) [11] as presented in Section 3.2. Another way to reduce the neural network energy consumption consists in reducing the input voltage of the embedded system [27] as discussed in Section 3.9. We review the main ideas and concepts from these previous studies in Sections 3.2, 3.3, 3.4 and 3.6. Next, we describe the contributions that were made during the PhD in Sections 3.5, 3.8 and 3.9. Notably, we introduce a critical comparison of all the different methods in Section 3.7. In particular, in the literature and most of methods discussed in this manuscript, the authors compare the accuracy and number of parameters of their method and the baseline. However, such a process gives only two points that cannot be used to perform a fair comparison. A good comparison would be to compare the accuracy for the same number of parameters and vice versa as discussed in Section 3.7.

3.2 Quantization

One of the most prominent approach in the field of compression of DNNs is quantization. In 2015, Courbariaux *et al.* introduce BinaryConnect (BC) [11] to binarize CNNs weights $\mathbf{W}$. This method constrains the weights to be either $+1$ or -1 during inference. As such,

Table 3.1: Comparison of obtained accuracy of full precision Alexnet, BC, BWN, BNN and XNOR-Net on ImageNet ILSVRC2012.

Full precision		BC		BWN		BNN		XNOR-Net	
Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
56.6%	80.2%	35.4%	61.0%	56.8%	79.4%	27.9%	50.42%	44.2%	69.2%

memory footprint is reduced and it is possible to replace all multiplication-accumulation operations by simple additions (or subtractions). BC uses the sign function to transform any real number to its binary quantized value ($+1$ or -1):

$$w^b = \begin{cases} 1 & \text{if } w \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

The method works as follows. During the training process, the inference is performed using the binary version of weights $\mathbf{W}^b$. However, the gradients are applied on the non-quantized values $\mathbf{W}$.

In [12] the same authors propose to extend this principle to activations. The proposed method is called Binary Neural Network (BNN). Introduced in [79], XNOR-Net is another method in which both weights and activations are binarized. The authors propose a method named Binary Weight Network (BWN) in which they attribute to each layer a scaling factor α_l and constrain weight values $\mathbf{W}_l$ to be either α_l or $-\alpha_l$, where $\alpha_l = \mathbf{E}(|\mathbf{W}_l|)$, and $\mathbf{W}_l^b = \alpha_l \times \mathbf{sign}(\mathbf{W}_l)$. They do similarly with the activations. The rest of the training process is performed the same way as for the BC method. Binarizing both weights and activations reduces memory and replaces multiplication-accumulation operations by XNOR operations followed by a bit-counting. In Table 3.1, we report the results from [79] showing that BWN achieves an accuracy comparable to the full precision network, significantly outperforming BC. It also shows that XNOR-Net achieves a better accuracy than BNN, and thus supports the fact adding a scaling factor is important to achieve a better accuracy.

In the same vein, Li *et al.* [57] propose Ternary Weight Networks (TWN) and introduce a third quantized value (0) to improve the accuracy. For each layer l, a symmetric threshold δ_l and a scaling factor α_l are used, and then weights are quantized into $\{-\alpha_l, 0, \alpha_l\}$ as follows:

Table 3.2: Comparaison of obtained accuracy of full precision ResNet-18, BWN, TWN and TTQ on ImageNet ILSVRC2012.

Full precision		BWN		TWN		TTQ	
Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
69.6%	89.2%	60.8%	83.0%	65.3%	82.6%	66.6%	83.6%

$$w_{l,i}^t = \begin{cases} \alpha_l & \text{if } w_{l,i} > \delta_l \\ 0 & \text{if } |w_{l,i}| \leq \delta_l \\ -\alpha_l & \text{if } w_{l,i} < -\delta_l, \end{cases}$$

where $\delta_l = 0.7 \times \mathbf{E}(\mathbf{W}_l)$, $\alpha_l = \mathbf{E}(\mathbf{W}_l^*)$ and $\mathbf{W}_l^* = \{w_{l,i}, |w_{l,i}| > \delta_l\}$.

In [120], the authors use Trained Ternary Quantization (TTQ) in which each layer l is associated with two scaling factors α_l^p and α_l^n for positive and negative weights, and a threshold δ_l, which are all used to quantize weight values $\mathbf{W}_l$ into $\{\alpha_l^n, 0, \alpha_l^n\}$. In addition, they propose to learn these scaling factors during the training phase. Table 3.2 shows that adding a third value and thus a second bit to quantize weights can significantly improve accuracy. We also observe that learning scaling factors is beneficial to the accuracy. Tables 3.2 and 3.1 show that it is more difficult to binarize small and optimized architectures such as ResNet than large and non optimized architectures such as AlexNet. Indeed, AlexNet is the first neural network used in the ImageNet challenge and generated a surge of interest in the field, but it is a large neural network architecture that may contain extra parameters, and thus its binarization (or quantization) is more easier.

These methods allow to scale down to 1 or 2 bits weight and activation values. However, the gradient and error values computed during backwards propagation as well as the weight updates are still using 32-bit Floating Point (32-FP) precision (*cf.* Figure 3.3: a). The reason is that gradient values $\mathbf{dW}$ can be much smaller than $\mathbf{W}$, thus a 32-FP is needed to perform the addition $\mathbf{dW} + \mathbf{W}$, and to achieve a good accuracy [73, 49] (*cf.* Figure 3.1). On the other hand, other approaches focus not only on quantizing weights and activations during inference, but also on gradients and errors during backward propagation. Micikevicius *et al.* [67] introduce Mixed Precision Training (MPT), in which they use IEEE Half precision 16-bit Floating Point format (16-HFP) (*cf.* Figure 3.3: b) to perform quantization. Note however that multiply-accumulate operations results are still encoded using 32-FP format. As shown in [67] and depicted in Figure 3.2, there are some values below minimum presentable range of 16-HFP that are set to 0 when quantizing, while a part of presentable range remains unused. Thus, the authors introduce a loss-scaling method to scale up gradients $\mathbf{dW}$ and $\mathbf{dX}$ and limit the number of values set

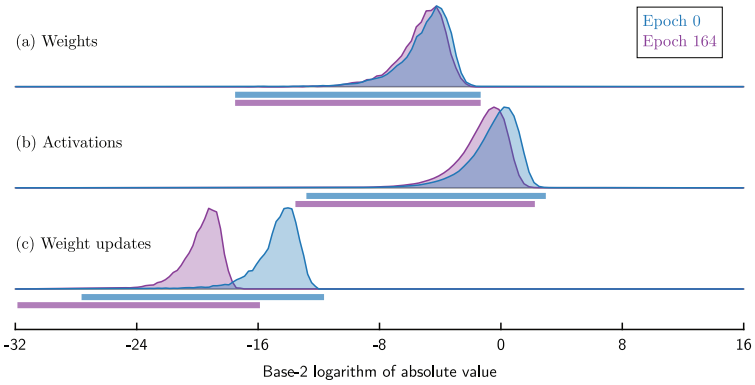


Figure 3.1: Overview of the distribution of values of weights, activations and gradient values (or weight updates) at the first training epoch (blue) and last training epoch (purple) of a ResNet architecture trained on CIFAR10. We see that gradient (or weight updates) values are way smaller than weights, in particular at the end of training. This figure is introduced in [49].

to 0 by using a larger part of 16-HFP presentable range. Dynamic Fixed Point (DFP) method [14] uses an unusual format for quantizing values, with a 16-bit mantissa and a shared exponent (*cf.* Figure 3.3: c), and a 32-FP format for results accumulation. Although these methods focus on quantizing weights and activations during inference, and gradients during back propagation, a 32-FP format is required for data accumulation. Moreover, full precision 32-FP representation is used to update weights.

A more recent work [101] proposes to train a DNN using 8-bit Floating Point (8-FP) quantizing format (*cf.* Figure 3.3: d) and a 16-bit Floating Point (16-FP) format for data accumulation. More precisely, the authors propose to use chunk based accumulation in which a long dot-product is divided into smaller equal size chunks. For each chunk, accumulation is performed to get a partial sum. Then, an accumulation of these partial sums is computed to get the final product value. The main idea is to add values of comparable magnitudes together and to avoid adding a large number of small ones, that would likely be considered as 0 in 8-FP. Table 3.3 compares and summarizes all the methods introduced in this subsection. Table 3.3 shows that it is harder to binarize activations than weight, and the accuracy drop is less significant when considering AlexNet, since it is a large and non optimized neural network architecture. Moreover, quantization methods need higher precision during training to perform well. Note that in Table 3.3 the CNN baseline used to evaluate both TWN and TTQ is Resnet-18, and to evaluate all other methods is AlexNet.

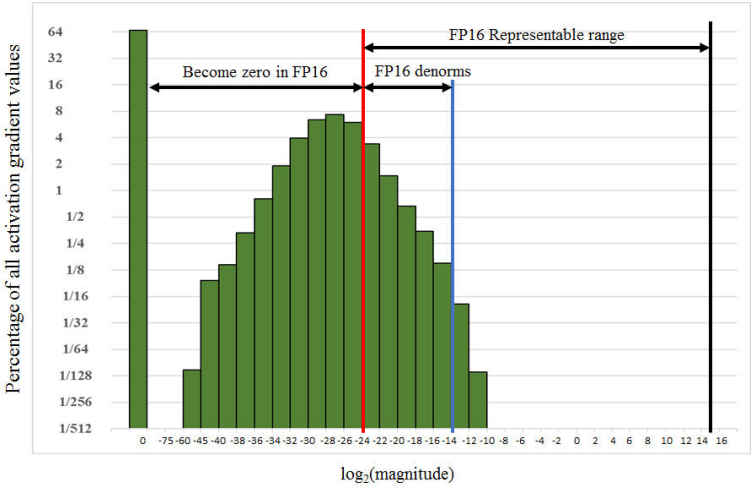


Figure 3.2: Histogram of activation gradient values during the training phase of Multibox SSD network [60] collected across all layers during 32-FP training. This figure was originally introduced in [67].

3.3 Pruning

In deep learning, a pruning-based method is a method that eliminates some neurons or connections according to a defined criterion in order to reduce the size of the neural network. Such a method evaluates the importance of each neuron, prunes the less important neurons and then finely tunes (*i.e.* retrains) the network. This concept has generated a lot of interest. For instance, Li *et al.* [56] use the absolute sum $\sum |\mathbf{W}_{l,i,:,:,:}|$ to measure the importance of a filter $\mathbf{W}_{l,i}$, then prune m filters with the smallest sum values and their corresponding output feature maps. Kernels in the next layer that are applied to pruned feature maps should also be removed since they are not used to compute the next output feature map (*cf.* Figure 3.4). Luo *et al.* define ThiNet [62], a pruning method which uses the importance of each feature map in layer $l+1$ to prune filters in layer l. Unlike in [56], where the importance of a filter (the operator) is used to decide which feature map is pruned, ThiNet uses the importance of the output feature map (which represents the input feature map of the next layer) to prune this feature map and its corresponding filter. The idea is to try to approximate the output of layer $l+1$ when using only a subset of input feature maps, and thus the non used input feature maps can be pruned. Each input feature map in layer $l+1$ is computed using one filter in layer l, hence when an input feature map is removed, the corresponding filter in layer l can be pruned. Moreover, and as shown in [56], kernels in layer $l+1$ that are applied to the pruned feature maps are also removed. Finally, fine tuning is applied to recover the neural network accuracy (*cf.* Figure 3.5).

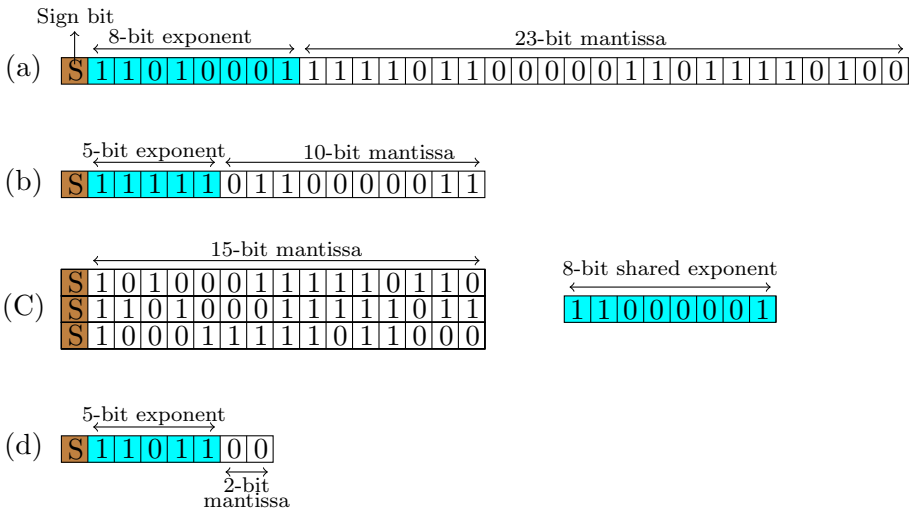


Figure 3.3: Overview of the precision of a) IEEE-754 floating point (32-FP), b) IEEE-754 half-floating point (16-HFP), c) dynamic fixed point (16-DFP), and d) 8 bit floating point (8-FP) data formats.

Table 3.3: Comparison of obtained top-1 accuracy on ImageNet ILSVRC2012 of full precision baselines and different quantization methods. Here "Acc" refers to accumulation, "Qan" to quantization method and "Net" to network.

Method	Bit precision					Top-1 accuracy (%)		Net
	W	**X**	**dW**	**dX**	Acc	Baseline	Qan	
BC [11]	1	32	32	32	32	56.6	35.4	AlexeNet
BNN [12]	1	1	32	32	32	56.6	27.9	AlexeNet
BWN [79]	1	32	32	32	32	56.6	56.8	AlexeNet
XNOR-Net [79]	1	1	32	32	32	56.6	44.2	AlexeNet
TWN [57]	2	32	32	32	32	69.6	65.3	Resnet-18
TTQ [120]	2	32	32	32	32	69.6	66.6	Resnet-18
DFP [14]	16	16	16	16	32	57.4	56.9	AlexeNet
MPT [67]	16	16	16	16	32	56.8	56.9	AlexeNet
8-FP training [101]	8	8	8	8	16	58.0	57.5	AlexeNet

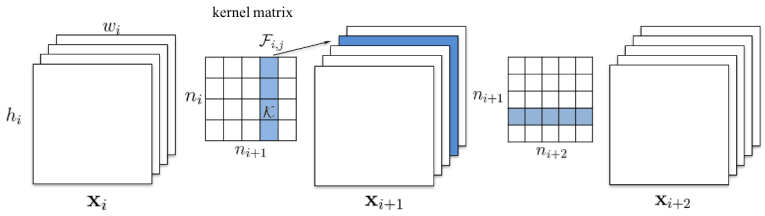


Figure 3.4: Overview of a filter pruning method. When a filter is pruned, its corresponding feature map and related kernels in the next layer are also removed. Note that this figure was originally introduced in [56].

Yu *et al.* [111] focus on applying Infinite Feature Selection (Inf-FS) [83], a feature ranking algorithm on the last DNN layer to obtain the importance score of each neuron. These importance scores are then propagated through the neural network to obtain the importance score of each neuron in each layer. The bottom ranked neurons are pruned, their score importance are not propagated and the network is fine-tuned to reduce accuracy drop (*cf.* Figure 3.6).

In [32], the authors introduce AutoML for Model Compression (AMC), a reinforcement learning based method to perform channel pruning. This method uses a trainable reinforcement learning agent which takes as input an embedding E_l from layer l, and outputs a sparsity ratio SR_l corresponding to channel pruning ratio in layer l. Then using SR_l, the layer l is compressed and layer $l+1$ is processed. Finally, a reward $R = -error * log(FLOPs)$ is computed and returned to the reinforcement learning agent. Note that FLOPs represents the total number of multiplication-addition required by a neural network to process data.

Yamamoto *et al.* [107] introduce Pruning Channels with Attention Statistics (PCAS), a pruning method which uses a channel pruning technique based on attention statistics by adding attention blocks to each layer. Starting from a pre-trained neural network, the authors add for each layer l an attention block which receives feature map $\mathbf{X}_l$ and outputs $\mathbf{SV}_l$, a scaling C_l dimensional vector. These attention blocks are trained without updating the parameters of the pre-trained network, and then for each layer l, a channel c is pruned if its corresponding scaling value $SV_{l,c}$ is lower than a defined threshold. Table 3.4 aims at resuming and comparing different pruning methods introduced in this subsection. It shows that it the accuracy drop is more significant when considering more complex datasets. Moreover, such results give only two points that cannot be used to fairly compare pruning methods with their corresponding baselines. In addition, such baselines can be improved when using the same hyper-parameters as pruning methods.

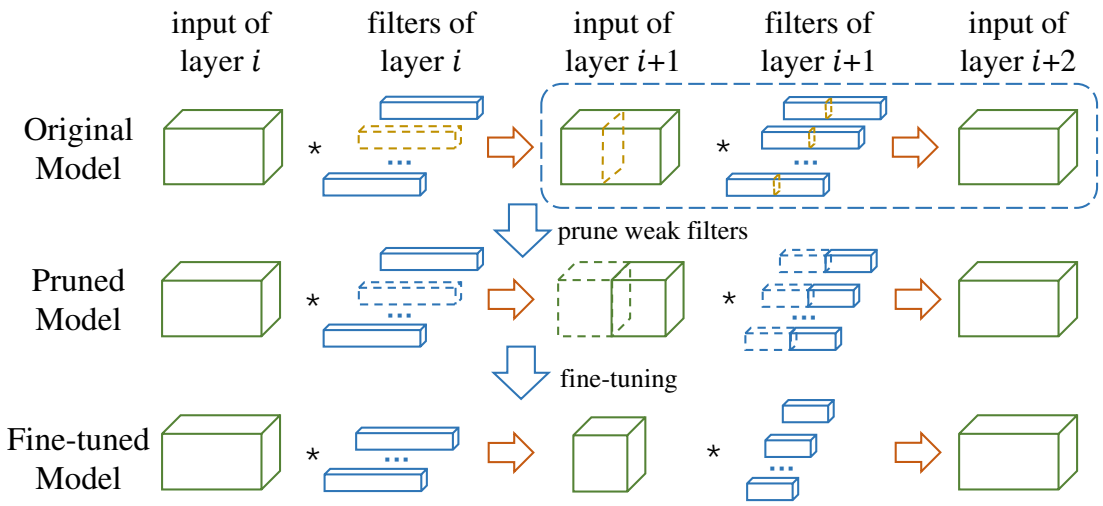


Figure 3.5: Overview of ThiNet method. First, on the first row are shown the least important input feature map of layer $l+1$, its corresponding kernels in the same layer, and the corresponding filter in layer l (dotted boxes). Then on the second row, all weak feature maps and their corresponding filters and kernels are removed. Finally on the third row, a fine tuning is applied on the pruned model to recover accuracy. Note that this figure was originally introduced in [62].

Table 3.4: Comparison of obtained top-1 accuracy, number of parameters (NP) and pruning ratio (PR) on CIFAR10 (C10), CIFAR100 (C100) and ImageNet ILSVRC2012 (ImNet) of different pruning methods applied on ResNet (RN) and MobileNet (M-Net).

Method	Network	Dataset	Baseline	Pruning	NP(M)	PR
Pruned-B [44]	RN-56	C10	93.04%	93.06%	0.73	13.7%
NISP [104]	RN-56	C10	93.04%	93.01%	0.47	42.6%
PCAS [107]	RN-56	C10	93.04%	93.58%	0.39	53.7%
AMC [32]	RN-50	C10	93.53%	93.55%	NA	60.0%
Pruned-B [44]	RN-50	C100	74.40%	73.60%	7.83	54.2%
PCAS [107]	RN-50	C100	74.66%	73.83%	4.02	76.5%
NISP [104]	RN-50	ImNet	72.68%	71.79%	14.36	33.7%
PCAS [107]	RN-50	ImNet	72.68%	72.64%	12.47	51.2%
Pruned-B [44]	RN-34	ImNet	73.23%	72.52%	20.10	7.2%
ThiNet [62]	RN-50	ImNet	72.88%	72.04%	16.94	33.7%
AMC [32]	M-NetV1	ImNet	70.90%	70.20%	13.20	34.3%

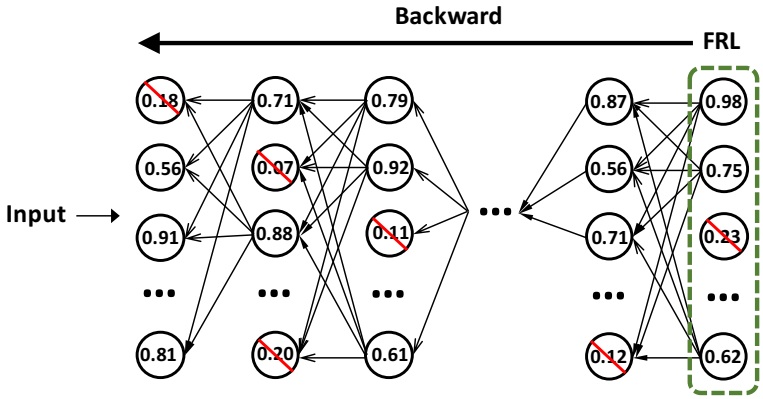


Figure 3.6: Overview of the propagation of neuron importance from the final response layer (FRL) to previous layers, while pruning neurons with low importance scores. Note that this figure was originally introduced in [111].

3.4 Light Architectures

Several authors have proposed to simplify neural network architectures in order to reduce the amount of computations, thus obtaining what we will refer to here as "light" architectures. One application domain of such architectures is mobile applications, for instance using trained networks on smartphones. In [40], the authors introduce SqueezeNet, an Alexnet accuracy level neural network with fewer parameters and a smaller model size. The authors build the CNN architecture using three main strategies. The first strategy is to replace the majority of 3×3 kernels by 1×1 kernels, since a 1×1 kernel has 9 times fewer parameters. The second one is to decrease the number of input channels of 3×3 kernels, since the total number of parameters of a convolutional layer l containing only 3×3 kernels is $9C_lC_{l+1}$, where C_l is number of input channels, and C_{l+1} is the number of output channels. The third strategy is to use downsampling only at the end of the network (on the last layers), so that convolutional layers handle large input feature maps which leads to higher accuracy as shown in [29]. To do so, the Fire module – a new building block – is introduced. A Fire module is made of a squeeze layer and an expand layer (*cf.* Figure 3.7). To fulfil the first strategy, Fire modules use more 1×1 than 3×3 kernels. In a Squeeze layer, the number of output channels is reduced, and then the number of input channels of expand layer which contains 3×3 kernels is also reduced, thus strategy 2 is fulfilled. Finally, the authors introduce max and average pooling layers and convolutional layers with stride higher than 1 deep in the network.

Howard *et al.* [37] propose MobileNet, a neural network architecture which uses a 3×3 depthwise convolution (3×3 DWConv) followed by 1×1 pointwise convolution (instead of 3×3 standard convolution) to reduce both the number of operations and

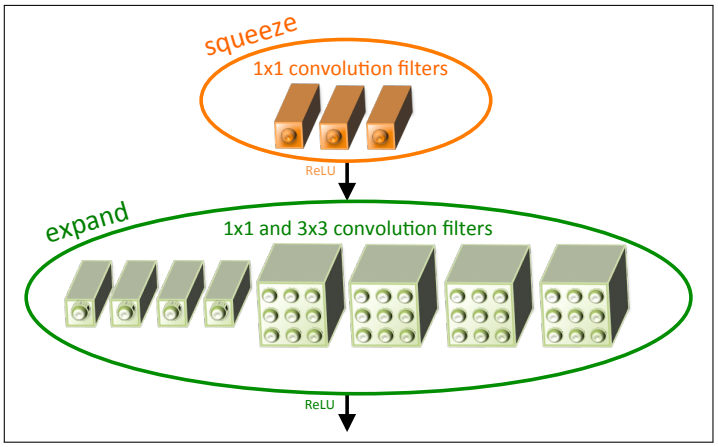
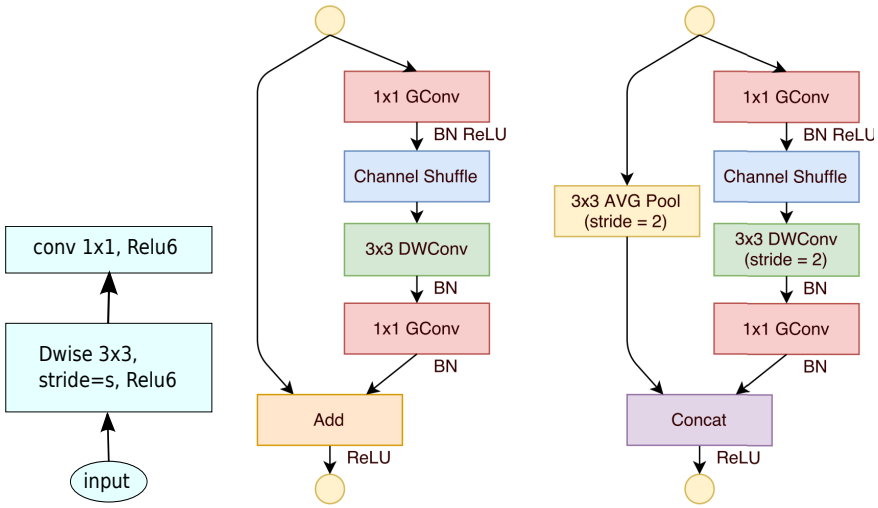


Figure 3.7: Overview of a Fire module. The first squeeze layer aims at reducing the number of input feature maps to 3×3 kernels to fulfil strategy 2, and the second expand layer aims at replacing some 3×3 kernels by 1×1 kernels to fulfil strategy 1. Note that this figure was originally introduced in [40].

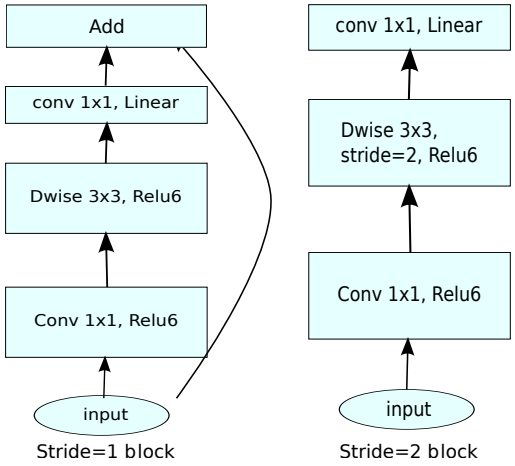
Table 3.5: Comparison of obtained top-1 accuracy and number of parameters on ImageNet ILSVRC2012 of SqueezeNet, MobileNetV1, MobileNetV2 and ShuffleNet.

Network	Accuracy(%)	Params (M)
SqueezeNet [40]	57.5	1.24
MobileNetV1 [37]	70.6	4.20
ShuffleNet [116]	71.5	3.40
MobileNetV2 [86]	72.0	3.40

parameters (*cf.* Figure 3.8 (a)). With the aim to improve MobileNet, Sandler *et al.* [86] come up with MobileNetV2 which uses a block containing 3 layers, a 1×1 convolution, a 3×3 DWConv and another 1×1 convolution. It also may use a residual connection that results in adding the input of the first 1×1 convolutional layer to the output of the second 1×1 convolutional layer (*cf.* Figure 3.8 (c)). In the same vein, Zhang *et al.* [116] use a channel shuffle concept, in which output channels of a grouped convolution (GConv) are randomly shuffled to define ShuffleNet Units, a key component to define the neural network architecture ShuffleNet (*cf.* Figure 3.8 (b)). Table 3.5 compares the obtained performance from SqueezeNet, MobileNet, MobileNetV2 and ShuffleNet, and the corresponding number of parameters.

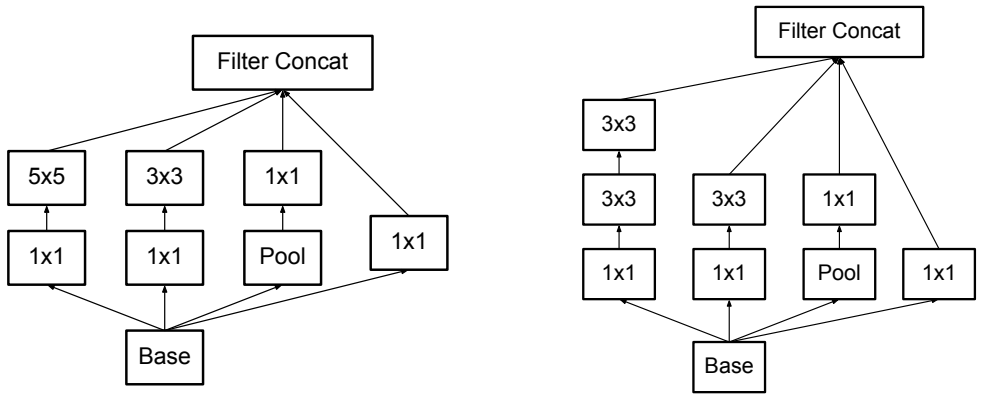


(a) MobileNetV1 (b) ShuffleNet



(c) MobileNetV2

Figure 3.8: Comparison of blocks for different architectures. Note that this figure was originally introduced in [86].



(a) InceptionV1 (b) InceptionV2

Figure 3.9: Comparison of blocks of InceptionV1 and InceptionV2. Note that this figure was originally introduced in [95].

3.5 Convolution Alternatives

In this subsection, we describe our contribution to the reduction of computations in CNNs, by introducing shift layers. The basic idea is to revisit convolution operations in order to save computations. In previous work, to reduce the number of neural network parameters, some methods focus on decomposing the convolution operation. For instance, Simonyan *et al.* [89] reduce the number of parameters of VGG by replacing 7×7 (resp. 5×5) convolutional layers by three (resp. two) 3×3 convolutional layers. Assuming that the number of both input and output channels is C, they use $3(9C^2) = 27C^2$ (resp. $2(9C^2) = 18C^2$) parameters instead of $49C^2$ (resp. $25C^2$). Moreover, they claim that they obtain a more discriminative decision function, since three (resp. two) non-linear activation functions are incorporated instead of one. To define a novel neural network architecture "InceptionV2" [95], the authors apply this method on the original inception module defined in [94] to improve the accuracy and reduce the number of parameters (*cf.* Figure 3.9: (a) and (b)). Moreover, they propose another alternative decomposition, in which a 7×7 convolution layer was replaced by a 1×7 convolution layer followed by a 7×1 convolution layer. As a consequence, there architecture uses only $2 \times 7C^2 = 14C^2$ parameters instead of $49C^2$.

Simultaneously, Wu *et al.* [104] and we [23] introduce Shift Layers (SLs), an alternative to Convolutional Layers (CLs). An SL consists in a shift operation to adjust data spatially, followed by a 1×1 convolution. To explain how a convolutional layer can be replaced by a shift layer, we consider a 1D convolutional case (other cases can easily be derived). Furthermore, and for simplicity reasons, we consider only one layer

l, and disregard downsampling and padding (*i.e.* border effects). For easy reading, we introduce the following notations: $C_l = C$, $C_{l+1} = D$, $H_l = H_{l+1} = H$, $S_l = S$, $\mathbf{X}_l = \mathbf{X}$, $\mathbf{X}_{l+1} = \mathbf{Y}$, $\mathbf{W}_l = \mathbf{W}$. Let us consider a 1D convolutional layer, and denote by $\mathbf{X} \in \mathbb{R}^{C \times H}$ the input feature map tensor, $\mathbf{W} \in \mathbb{R}^{D \times C \times S}$ the weight tensor, and $\mathbf{Y} \in \mathbb{R}^{D \times H}$ the output feature map tensor. The convolution operation is depicted in Figure 3.11: (1), and can be computed as follows:

$$y_{d,h} = \sum_{c=1}^{C} \sum_{h'=1}^{S} x_{c,h+h'-\lceil S/2 \rceil} w_{d,c,h'}, 1 \leq d \leq D, 1 \leq h \leq H \ . \tag{3.1}$$

Basically, to obtain a shift layer, for each kernel $\mathbf{W}_{d,c,\cdot,\cdot}, 1 \leq d \leq D, 1 \leq c \leq C$, we prune all weights but one, and end up with exactly one weight $w_{d,c,i_{d,c}}$ per kernel, where $i_{d,c}$ represents the index of non-pruned weight. Then Equation 3.1 becomes:

$$y_{d,h} = \sum_{c=1}^{C} x_{c,h+h_{d,c}-\lceil S/2 \rceil} w_{d,c,h_{d,c}} \tag{3.2}$$

$$= \sum_{c=1}^{C} \tilde{x}_{c,h} \tilde{w}_{d,c} \ , \tag{3.3}$$

where $\tilde{x}_{c,h} = x_{c,h+h_{d,c}-\lceil S/2 \rceil}$ and $\tilde{w}_{d,c} = w_{d,c,h_{d,c}}$. From Equation 3.3 and as shown in Figure 3.11, we observe that the convolutional operation is transformed into a shifted input feature map $\tilde{\mathbf{X}}$ convolved with a kernel of size 1. Thus, the convolution operation is replaced by a shift operation followed by a 1×1 convolution. To estimate the drop in performance caused by this pruning method, we randomly remove m weights per kernel and see the behaviour of the accuracy. We use CIFAR10, and compare various modern CNN architectures such as Resnet [30], Wide-Resnet [113], Densenet [39], and Mobilenet [86]. Note that these architectures contain 1×1 and 3×3 convolutional kernels only. Thus we apply the proposed method on the 3×3 kernels. Figure 3.10 shows that the accuracy of the architecture is quite robust to this process, even when 8 out of the 9 connections in slices of 3×3 kernels are randomly removed.

In this method, the shifts are hand-crafted and determined before the training process (*i.e.* we choose which weight we keep for each kernel at the initialisation, and before starting the training process). To improve the accuracy of the shift operation method, Jeon *et al.* [44] propose an active shift layer (ASL), to replace the hand-crafted shifts by learnable parameters which are optimised during back propagation. The authors formulate the shift value α_c (β_c can be defined when considering 2D convolution) corresponding to each feature map $\mathbf{X}_{c,\cdot}$ as a learnable parameter to define the amount of shift. The

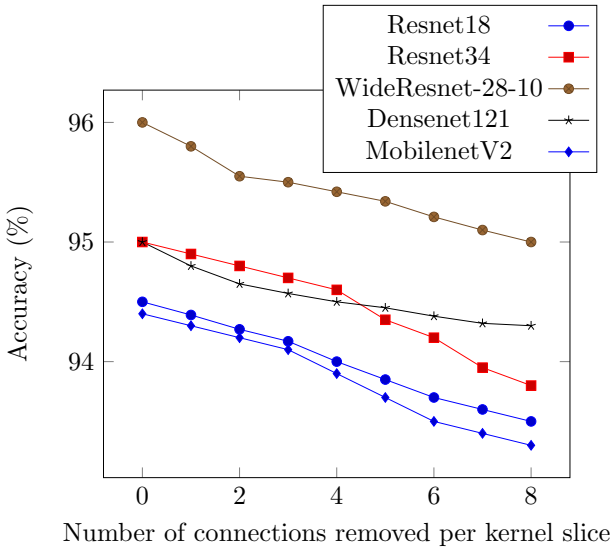


Figure 3.10: Evolution of accuracy as a function of the number of connections removed.

learnable parameter α_c should be a real number and not an integer, so it can be made differentiable and optimised. This is why the authors use bilinear interpolation [43] to define non-integer shift as follows:

$$\tilde{x}_{c,h+\alpha_c} = Z_c^1(1 - \Delta\alpha_c) + Z_c^2\Delta\alpha_c \ , \tag{3.4}$$

where $\Delta\alpha_c = \alpha_c - \lfloor\alpha_c\rfloor$, and Z_c^1 and Z_c^2 are the two nearest integer points used to compute bilinear interpolation as follows:

$$Z_c^1 = x_{c,h} + \lfloor\alpha_c\rfloor, \quad Z_c^2 = x_{c,h} + \lfloor\alpha_c\rfloor + 1 \ . \tag{3.5}$$

This method aims at avoiding accuracy drops caused by the hand-crafted shifts. However, to perform a shift operation during inference, ASL needs to compute a non-integer shift which can be computationally expensive compared to an integer shift where just a memory access is needed, and thus the result architecture requires to perform interpolations and does not fall into the original shift layer formulation. To furthermore improve this method, we propose Shift Attention Layer (SAL), a pruning-shift attention-based method [26]. SAL uses pruning in such a way to keep only one weight per kernel, and thus not only to reduce memory of CNNs, but also to replace convolutional layers by shift layers. The idea we propose is to add an attention mechanism to the convolution layer which aims at identifying which weights should be kept in each kernel. As such, we introduce $\mathbf{A} \in \mathbb{R}^{D\times C\times H}$ an attention tensor containing as many elements as weights in the weight tensor. Each value of $\mathbf{A}$ is normalised between 0 and 1 and represents

how important the corresponding weight in $\mathbf{W}$ is (*cf.* Figure 3.11: (3)). At the end of the training process, $\mathbf{A}$ becomes binary, with only one nonzero element per slice $\mathbf{A}_{d,c,\cdot}$, corresponding to the weights in $\mathbf{W}$ that should be kept.

More precisely, each slice $\mathbf{A}_{d,c,\cdot}$ is normalized using a softmax function with temperature T. The temperature is decreased smoothly along the training process. Such a method eventually finds out that the most accurate solution is the convolution itself, and puts all attention tensor elements to the same value $1/S$, thus it can still compute a convolution operation. To force the layer to select some of the weights, we divide each slice $\mathbf{A}_{d,c,\cdot}$ elements by their standard deviation (sd) before applying the softmax, so we end up with $sd = 1$ and then prevent the elements from converging to the same value. Algorithm 1 summarises the training process of one layer. At the end of the training, the selected weight in each kernel $\mathbf{W}_{d,c,\cdot}$ corresponds to the maximum value in $\mathbf{A}_{d,c,\cdot}$.

Algorithm 1 SAL algorithm of one layer

Inputs: Input tensor $\mathbf{X}$,
Initial softmax temperature T, Constant $\alpha < 1$.

for each training iteration **do**
  $T = \alpha T$
  for $d := 1$ to D **do**
    for $c := 1$ to C **do**
      $\mathbf{A}_{d,c,\cdot} = \frac{\mathbf{A}_{d,c,\cdot}}{sd(\mathbf{A}_{d,c,\cdot})}$
      $\mathbf{A}_{d,c,\cdot} = Softmax(\mathbf{A}_{d,c,\cdot}, T)$
    end for
  end for
  $\mathbf{W}_A = \mathbf{W} \cdot \mathbf{A}$ ($\cdot$ is the pointwise multiplication)
  Compute standard convolution as described in Equation 3.1 using input tensor $\mathbf{X}$ and weight tensor $\mathbf{W}_A$ instead of $\mathbf{W}$.
  Update $\mathbf{W}$ and $\mathbf{A}$ via back-propagation.
end for

To evaluate the performance of the proposed SAL method, we adopt a benchmark protocol that compares the obtained performance with CNNs baseline, vanilla shift layers and other pruning methods.

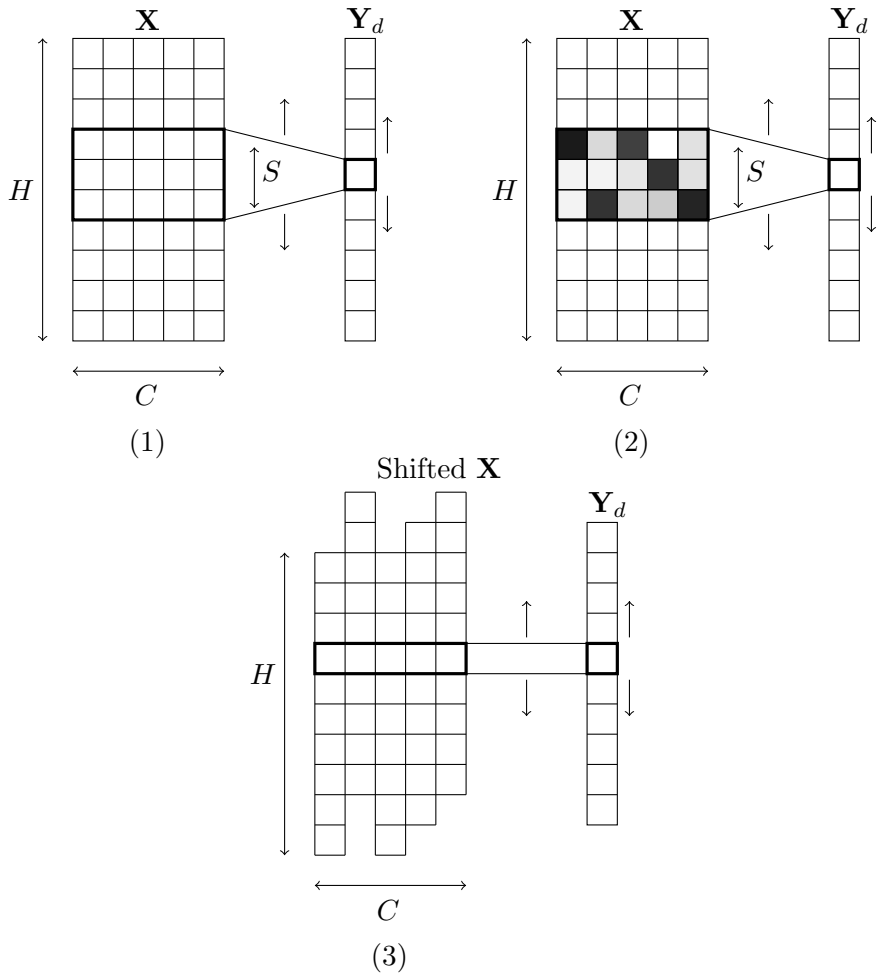


Figure 3.11: An overview of the proposed method: we depict here the computation for a single output feature map d. Panel (1) represents a standard convolutional operation: the weight filter $\mathbf{W}_{d,\cdot,\cdot}$ containing SH weights is moved along the spatial dimension of the input to produce each output in $\mathbf{Y}_d$. In panel (2), we depict the attention tensor $\mathbf{A}$ on top of the weight filter: the darker the cell is, the more important the corresponding weight has been identified to be. At the end of the training process, $\mathbf{A}$ should contain only binary values with a single 1 per slice $\mathbf{A}_{d,c,\cdot}$. In panel (3), we depict the corresponding obtained shift layer: for each slice along the input feature maps, the cell with the highest attention is kept and the others are disregarded. As a consequence, the initial convolution with a kernel size S has been replaced by a convolution with a kernel size 1 on a shifted version of the input $\mathbf{X}$.

Benchmark Protocol

We perform the evaluation on three vision datasets: CIFAR10, CIFAR100 and ImageNet ILSVRC 2012. We test Resnet-20/56 on CIFAR10 and Resnet-20/50 on CIFAR100 using the following training hyper-parameters: we use 300 epochs to train Resnet-20 and 400 to train Resnet-56/50, 0.1 as initial learning rate and divided it by 10 after each 100 epochs, a training batch of 128 examples, the initial/final softmax temperatures are 6.7/0.02, and the temperature is multiplied at each step (each time a batch of 128 examples is processed) by $\alpha = 0.99994, 0.99996$ when using 300, 400 epochs respectively.

We test Resnet-w32 and Resnet-w64 defined in [44] on ImageNet ILSVRC 2012 using the following training hyper-parameters: 90 epochs to train the neural networks, a batch of 1024 examples, 0.1 as initial learning rate that is divided by 10 after each 30 epochs, initial/final softmax temperatures are 6.7/0.016 so that the temperature update at each step is $\alpha = 0.99995$. We also used standard data augmentation defined in [51]. Note that these latest parameters were chosen because they perform well in practice.

Let us point out that the values of temperatures were obtained by using a grid search. The fact the final temperature is not zero means that the tensors $\mathbf{A}$ may contain nonbinary values. This is why we binarize $\mathbf{A}$ using a hard max to obtain the corresponding shift layers before evaluating on the test set.

Results

SAL is a pruning method aiming at reducing memory and number of operations, and also at replacing convolutional layers by shift layers. Hence for a fair evaluation we need to compare it to shift-based module methods such as SL ans ASL, but also to pruning methods described in Section 3.3.

To compare SAL with shift-based module methods (*cf.* Table 3.6), and pruning methods (*cf.* Table 3.7 and Table 3.8), we perform experiments on CIFAR10 and CIFAR100. Table 3.6 shows that our method achieves a better accuracy with fewer parameters than the baseline and other shift-module based method. Tables 3.7 and 3.8 show that SAL is comparable or better in term of accuracy and number of parameters/floating point operations (FLOPs) when compared with other pruning methods.

In the second experiment, an average of $\mathbf{A}$ along channel dimension is plotted at the end of training process to show the proportion of each kept position in slices $\mathbf{A}_{d,c,\cdot,\cdot}$. Figure 3.12 plots a heat-map to represent the proportion of kept weights through

Table 3.6: Comparison of accuracy and number of parameters between the baseline CNN architecture (ResNet20), vanilla SL, ASL, and SAL (the proposed method) on both CIFAR10 and CIFAR100.

		CIFAR10		CIFAR100	
		Accuracy	Params	Accuracy	Params
CLs	Baseline	94.66%	1.22 M	73.7%	1.24 M
SLs	Vanilla SL [104]	93.17%	1.2 M	72.56%	1.23 M
	SAL (ours)	**95.52**%	**0.98** M	**77.39**%	**1.01** M
Interpolate	ASL [44]	94.53%	0.99 M	76.73%	1.02 M

Table 3.7: Comparison of accuracy, number of parameters and number of floating point operations (FLOPs) between baseline architecture (Resnet-56), SAL (the proposed method), and some other pruning methods on CIFAR10. Note that the number between () refers to the result obtained by the baseline used for each method.

		CIFAR10		
		Accuracy	Params (M)	FLOPs (M)
Pruning	Pruned-B [56]	93.06%(93.04)	0.73(0.85)	91(126)
	NISP [111]	93.01%(93.04)	0.49(0.85)	71(126)
	PCAS [107]	93.58%(93.04)	0.39(0.85)	56(126)
	SAL (ours)	**94**%(93.04)	**0.36**(0.85)	**42**(126)

Table 3.8: Comparison of accuracy, number of parameters and number of floating point operations (FLOPs) between baseline architecture (Resnet-50), SAL (the proposed method), and some other pruning methods on CIFAR100. Note that the number between () refers to the result obtained by the baseline used for each method.

		CIFAR100		
		Accuracy	Params (M)	FLOPs (M)
Pruning	Pruned-B [56]	73.6%(74.46)	7.83(17.1)	616(1409)
	PCAS [107]	73.84%(74.46)	4.02(17.1)	475(1409)
	SAL (ours)	**77.6**%(78)	**3.9** (16.9)	**251**(1308)

$\mathbf{W}_{d,c,\cdot,\cdot}, \forall d, c$, for the 4 first CLs (first row), and the 4 last CLs (second row), of Resnet-20 trained on CIFAR10, and where attention tensors $\mathbf{A}$ values are initialised uniformly at random. An interesting thing to notice is that at the end of training, first layers present a uniform distribution of kept weight, while last layers show an asymmetric distribution in which most of kept weights are in corner positions. This interestingly suggests that shift-layers would benefit from a non regular number of shifts in each direction.

To see how much kept weight positions at the end of training depend on the initialisation, we propose to perform an other initialisation where $\mathbf{A}$ values are initialised uniformly at random except the centre value $\mathbf{A}_{d,c,\lfloor S/2 \rfloor,\lfloor S/2 \rfloor}$ to which we attribute the maximum over the corresponding slice $\max(\mathbf{A}_{d,c,\cdot,\cdot})$. Figure 3.13 shows that almost all kept weights in the first layer are slices centres. In the intermediate layers, we see a uniform distribution of kept weight positions, and we observe the same phenomenon in last layers as in the previous experiment. This shows that the uniform distribution of kept weight positions in first layers is not caused by the initialisation of $\mathbf{A}$. We also plot a heat-map of kept weight positions distribution of ResNet-56 trained on CIFAR10, and where $\mathbf{A}$ is initialised uniformly at random. Figure 3.14 shows that for the first layers, the number of kept weights is more important on the centre row than at other positions. However, we see on the last layers that there is more kept weights in the corners than at other positions, just as seen for previous experiments.

For further results, we run an experiment in which we replace all 3×3 Resnet-20 kernels by 5×5 kernels, and train the network on CIFAR10. We observe in Figure 3.15 that the weights of the centre in first layers are more important than at other positions. We also see that on the last layers the weight distribution is still not uniform, and the weights on the corners are more important in the last layer.

From all these experiments, we consistently observe that in deeper layers, the method tends to keep more weights in corner positions than others, and this independently from initialization process or neural network architecture. This observation interestingly questions the hyper-parameters used by the corresponding architectures. It clearly seems the network is more interested in locality in the initial layers than it is in the last layers. Based on this finding, we modified the vanilla shift layer method, using an equivalent uneven distribution of shifts as the one found in our experiments. As such, shifts are predetermined but not uniform. We obtained an accuracy of 94.8% on Resnet-20 and CIFAR10, to be compared to the 93.17% accuracy from Table 3.6. Interestingly, this accuracy is even better than the results obtained using the method in [44]. On the other hand, the obtained accuracy remains lower than that of SAL, suggesting that selecting the shifts during the learning process is still more efficient than

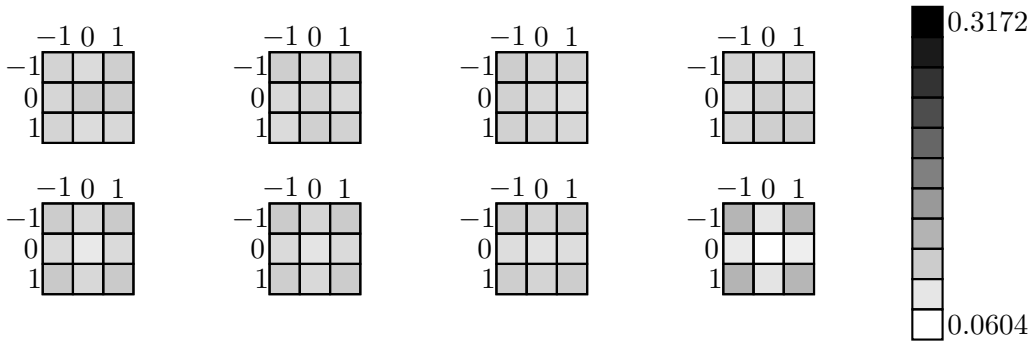


Figure 3.12: Heat maps representing the average values in **A** for various layers in the Resnet-20 architecture trained on CIFAR10. In this experiment, values in **A** are initialized uniformly at random. The first row represents the 4 first layers and the second row the 4 last layers of Resnet-20.

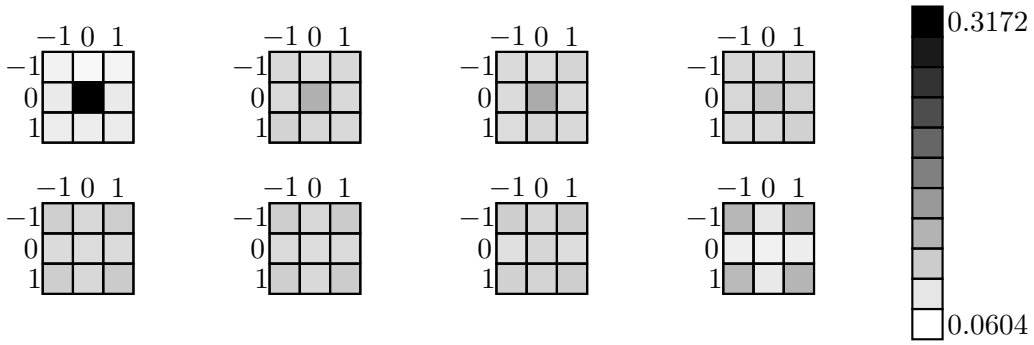


Figure 3.13: Heat maps representing the average values in **A** for various layers in the Resnet-20 architecture trained on CIFAR10. In this experiment, values in **A** are initialized uniformly at random but the centre value that takes the maximum over the corresponding slice. The first row represents the 4 first layers and the second row the 4 last layers of Resnet-20.

having a good choice of predetermined shift proportions.

In a third experiment, we observe the effect of initial and final temperature choices on accuracy. Figure 3.16: left represents the evolution of accuracy of Resnet-20/56 trained on CIFAR10 and Resnet-20/50 trained on CIFAR100 as function of final temperature while initial temperature is fixed at 6.7. It shows that the accuracy decreases when the final temperature becomes too high. Note that when the final temperature is large, obtained values in **A** at the end of the training process can be far from binary. In all cases, we round the values in **A** to the nearest integer before computing the accuracy. This experiment shows that final temperature values need to be small enough so the softmax can push the highest value to 1 and the other values to 0. Figure 3.16: right shows

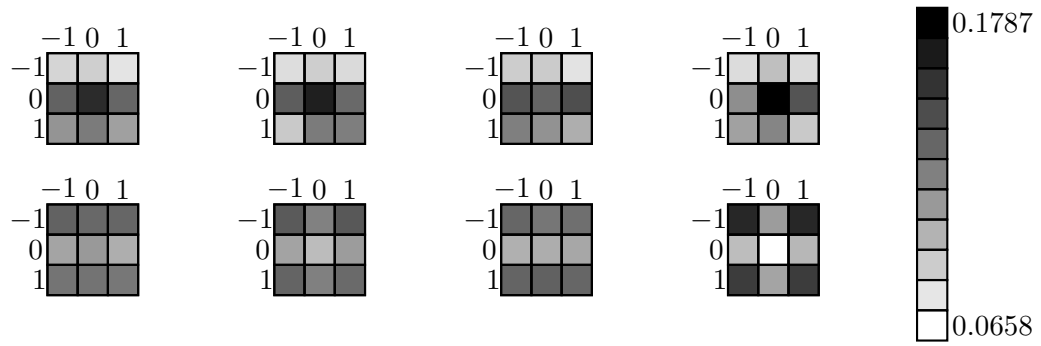


Figure 3.14: Heat maps representing the average values in **A** for various layers in the Resnet-56 architecture trained on CIFAR10. In this experiment, values in **A** are initialized uniformly at random. The first row represents the 4 first layers and the second row the 4 last layers of Resnet-56.

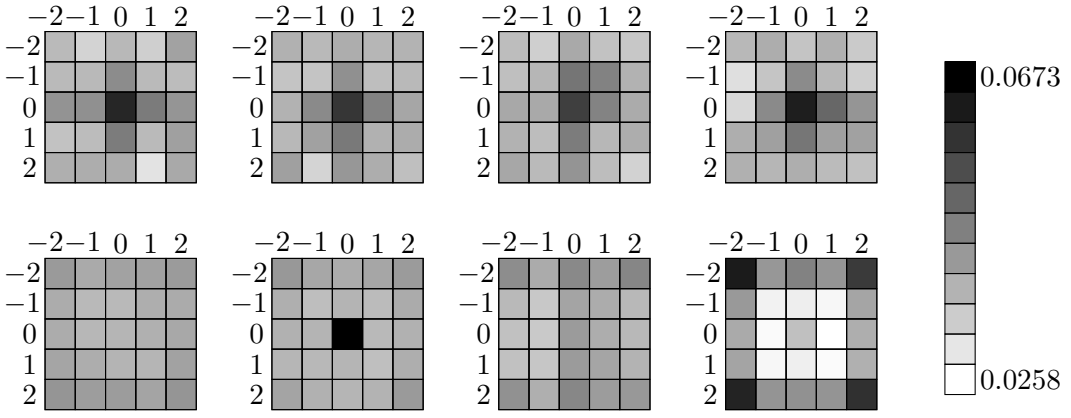


Figure 3.15: Heat maps representing the average values in **A** for various layers in the Resnet-20 architecture with 5×5 kernels trained on CIFAR10. In this experiment, values in **A** are initialized uniformly at random. The first row represents the 4 first layers and the second row the 4 last layers.

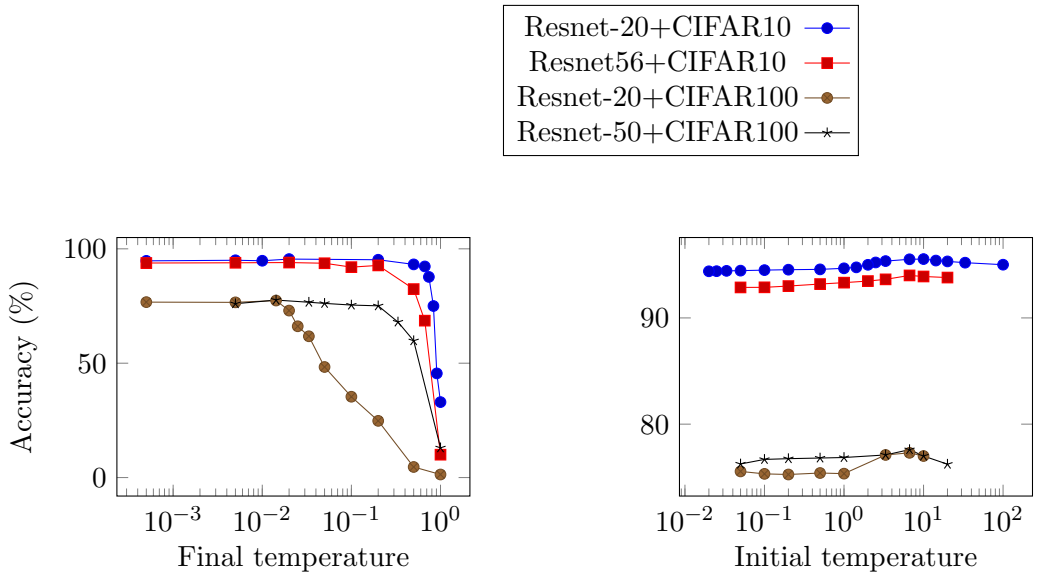


Figure 3.16: Evolution of accuracy of Resnet-20/56 trained on CIFAR10 and Resnet-20/50 trained on CIFAR100 as function of final temperature (left), and as function of initial temperature (right).

the behaviour of the accuracy of Resnet-20/56 trained on CIFAR10 and Resnet-20/50 trained on CIFAR100 when initial temperature is changed and final temperature is fixed at 0.02. We see an interesting region between 10 and 6.7 in which the accuracy is better. It is worth mentioning that the choice of initial and final temperatures is sensitive with respect to the obtained accuracy. Throughout our experiments, we observed that a too slow decrease in temperature causes the architecture to get stuck in local minima that are poorly fitted to the ending rounding operation. On the contrary, a too fast decrease in temperature prevents the learning procedure from finding the best shifts and boils down to an accuracy that is very similar to that of vanilla shift layers.

In the fourth experiment, we compare the accuracy, memory usage and FLOPs of SAL against vanilla Shiftnet and standard CNN on ImageNet ILSVRC 2012. Table 3.9 shows that SAL is able to obtain better accuracies than vanilla Shiftnet and standard CNN for the same memory and FLOPs budget.

3.6 Other Methods

To reduce CNNs memory footprint, other works propose to investigate other leads as weights sharing, or encoding information theory based techniques. Searching other meth-

Table 3.9: Comparison of accuracy, number of parameters and FLOPs between a standard CNN, SAL and vanilla Shiftnet on ImageNet ILSVRC 2012.

		Top-1	Top-5	Params	FLOPs
Large budget	Resnet-w24 (CLs)	63.47%	85.52%	**3.2** M	664 M
	Shiftnet-A [104]	70.1%	89.7%	4.1 M	1.4G
	Resnet-w64 + SAL (ours)	**71**%	**89.8**%	3.3 M	**538** M
Small budget	Resnet-w16 (CLs)	56.6%	80.4%	1.4 M	295 M
	Shiftnet-B [104]	61.2%	83.6%	1.1 M	371 M
	Resnet-w32 + SAL (ours)	**62.7**%	**84**%	**0.97** M	**136** M

ods and techniques to reduce complexity and memory footprint of CNNs can be relevant in such a way some different methods can be combined in order to further compress CNN models. For instance, Gong *et al.* [**?**] use vector quantization to compress DNNs size while keeping an accuracy comparable to the state-of-the-art. However, the authors compress only the fully connected layers, and ignore the convolutional layers. Han *et al.* [28] present deep compression, a quantization method built upon three main stages to reduce the storage required by neural network, while preserving the accuracy (*cf.* Figure 3.17). The authors propose to start by a pruning stage, where all connections with weight values below a defined threshold are pruned and removed from the network. To keep a good accuracy after the pruning process, they retrain the network to learn the new weight values for the new sparse architecture. They claim that pruning stage could divide the number of parameters by 9 (resp. 16) for Alexnet (resp. VGG-16). Then, a weight sharing stage is applied on the resulting sparse neural network architecture to further compress the network by reducing the number of bits required to store weight values. For this purpose, the authors use k-means clustering to identify which weight falls into which cluster, and thus all weights belonging to the same cluster are replaced by the same value corresponding to the centroid of the cluster. A fine-tuning process is computed after the clustering stage to keep a good accuracy. During back propagation, weight gradients of the same cluster are summed, and the resulting values are used to update the centroid values (*cf.* Figure 3.18). At this stage, the authors claim that they divide the number of parameters by 27 (resp. 31) for Alexnet (resp. VGG-16). Finally, they apply Huffman coding to take advantage of the weight values distribution. At the end, the authors show that they divide the number of parameters by 35 (resp. 49) for Alexnet (resp. VGG-16). This method is also applied to SqueezeNet, a neural network architecture introduced in [40] and defined in Section 3.4, allowing Squeezenet to achieve

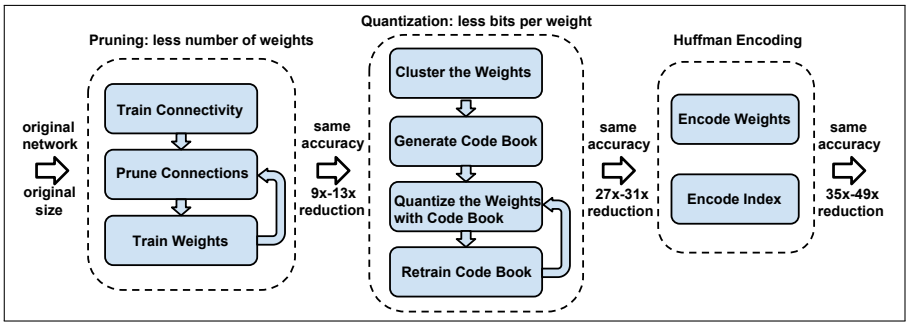


Figure 3.17: Overview of deep compression method. this method contains three compression stages: a pruning based method to compress original network by a factor between 9× and 13×, a weight sharing method based on k-means to further compress the network by a factor between 27× and 31× and a Huffman coding. At the end the neural network is compressed by a factor between 35× and 49× while keeping the same accuracy as the original one. Note that this figure was originally introduced in [28].

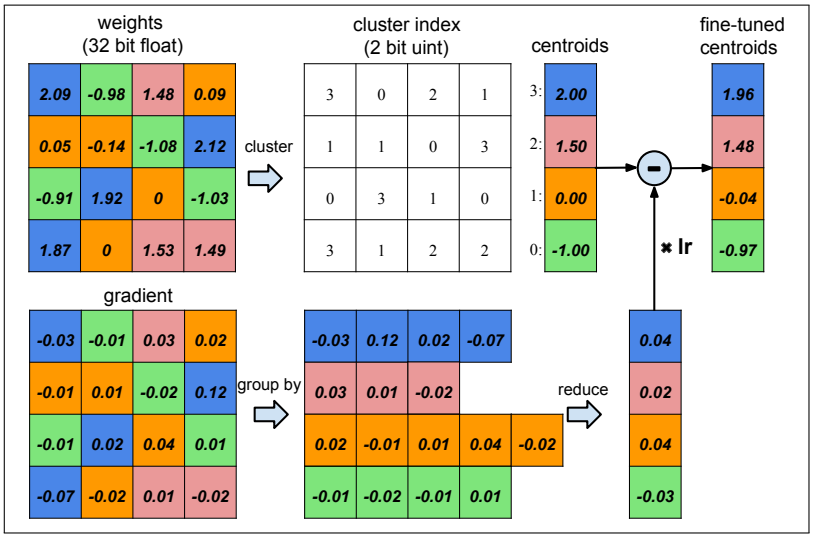


Figure 3.18: Overview of Weight sharing quantization method (top) and fine tuning process (bottom). Note that this figure was originally introduced in [28].

an AlxeNet accuracy on ImageNet ILSVRC2012 using 1/50× as many parameters and less than $0.5MB$ of memory.

The weight sharing method introduced above uses k-means, and thus it assigns weights to clusters once and for all at one step in the training process. This sudden factorisation can lead to drop in accuracy. To alleviate this drawback, Wu *et al.* [105] propose deep k-means, a weight sharing method based on spectrally relaxed k-means regularisation introduced in [114], and defined by Equation 3.6, where Tr denotes the matrix trace, and considering n_j the number of weights belonging to cluster j. Note

Table 3.10: Comparison of obtained top-1 accuracy, and compression ratio (CR) when using deep compression (DC) and deep k-means (DK).

Method	Network	Dataset	Baseline	Compressed	CR
DC [28]	Alexnet	ImageNet	57.20%	57.20%	35×
DC [28]	VGG-16	ImageNet	68.5%	68.83%	49×
DC [28]	SqueezeNet	ImageNet	57.50%	57.50%	10.2×
DK [105]	WideResNet	CIFAR10	93.52%	89.03%	50×
DK [105]	GoogLeNet	ImageNet	69.76%	67.81%	4×

that $\mathbf{B}$ would be a matrix such as $B_{ij} = 1/\sqrt{n_j}$ if column i belongs to the cluster j and $B_{ij} = 0$ otherwise, and $\mathbf{B}^T\mathbf{B} = I$.

$$\min_{\mathbf{W};\mathbf{B}} Tr(\mathbf{W}^T\mathbf{W}) - Tr(\mathbf{B}^T\mathbf{W}^T\mathbf{W}\mathbf{B}). \tag{3.6}$$

This regularisation allows to learn the assignments of neural network weights during the retraining (or fine-tuning) process , and thus the cost function minimised by retraining process becomes (where λ is a scalar):

$$\min_{\mathbf{W},\mathbf{B}} E(\mathbf{W}) + \frac{\lambda}{2}[Tr(\mathbf{W}^T\mathbf{W}) - Tr(\mathbf{B}^T\mathbf{W}^T\mathbf{W}\mathbf{B})]. \tag{3.7}$$

After retraining, a k-means is performed to assign weights to clusters. Table 3.10 compares the accuracy obtained when such compressing methods are used.

There is also other methods based on distillation that aim at reducing neural networks memory footprint by transferring the knowledge from a bigger model to a smaller one [34, 48], and even combine it with other compressing methods to further reduce neural networks size [91].

3.7 Comparison and Combination of Different Compression Methods

As described above, there are different compression methods that aim at reducing DNNs size. A relevant question would be: which method fits better in a specific limited resources embedded system when a specific accuracy drop is allowed? Moreover, how can

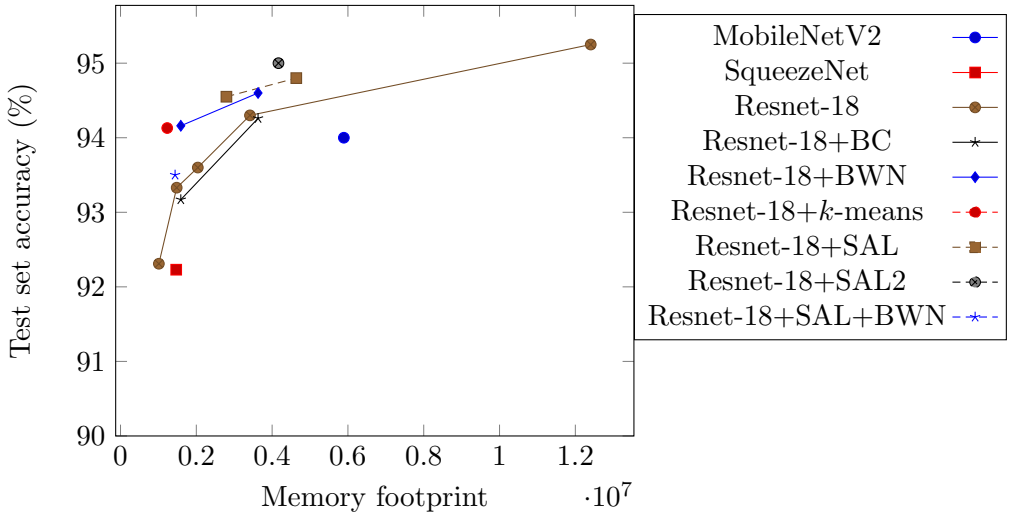


Figure 3.19: Comparison of accuracy when applying compression methods on a CNN baseline (Resnet-18) and other different CNN architectures.

these methods be combined in order to further compress DNNs when achieving an acceptable accuracy? We propose to evaluate in Figure 3.19 the compression methods and their combinations, and determinate which method or combination of methods gives a considerable compression rate while keeping a good accuracy. To get a good approximation of memory needed to implement a neural network on an embedded system, we consider both weights and activations when one input data is processes, thus memory footprint of a DNN will be the memory needed to store both weights and activations. We do not consider pruning methods since authors only present weight compression ratio in their contributions, which cannot be used to determinate activation compression ratio.

In our evaluation we compare SAL with Binary Connect (BC) [11], Binary Weight Network (BWN) [79] and k-means [28] applied to Resnet-18, and with MobileNetV2 [86] and Squeezenet [40]. We also perform the comparison with another version of SAL denoted SAL2, in which we keep two weights per kernel instead of one, and with a combination of SAL and BWN. We use different versions of Resnet-18 with different number of weights and activations as baseline. We also apply compression methods on these different versions in order to compare the accuracy obtained by the different methods for the same budget of memory. Figure 3.19 shows that the baseline outperforms MobileNetV2, Squeezenet and applying BC on Resnet-18. It also shows that SAL and SAL2 outperform all other methods, and SAL2 achieves a better accuracy than SAL.

Table 3.11: Comparison of accuracy and memory usage between Resnet-20 baseline, SAL, SAL with BC and SAL with BWN on CIFAR10.

	Accuracy(%)	Memory usage (Mb)
baseline	94.66	39.04
SAL	95.52	31.36
SAL + BC	93.20	6.87
SAL + BWN	94.00	6.87

3.8 Hardware Implementation

As depicted in Section 3.5, SAL is an efficient pruning method that reduces both memory and computations. Moreover, it replaces the standard and complex convolutional by a simple shift operation followed by 1×1 convolution. In [23], we propose to combine a shift based module method with BC [11], and end up with one kept binary weight per kernel. Consequently, we replace the convolutional operation by only a low-cost multiplexer, and propose an efficient hardware architecture to implement such a method on FPGA. Such a combination still achieves a comparable accuracy to state-of-the-art while using less parameters as depicted in Table 3.11.

In this section, we introduce the hardware architecture of SAL combined with BC or BWN, its different components, and the way they are connected. Then, we present the hardware implementation of the proposed combination, applied to Resnet-18, on FPGA. Since that the same scaling factor α is used to define all the weights of the same layer, BWN can be assimilated to BC with one multiplication at the final stage by α. Thus, the same hardware architecture can be used to implement the combination of SAL with BC and BWN on FPGA. Note that for simplicity reasons, we use the following notations: $\mathbf{X}_l = \mathbf{X}$, $\mathbf{X}_{l+1} = \mathbf{Y}$, $\mathbf{W}_l = \mathbf{W}$, $C_l = C$, $C_{l+1} = D$, $H_l = H_{l+1} = H$, $R_l = R$, $R_{l+1} = R'$.

3.8.1 Hardware Architecture

In Figure 3.20, we depict the proposed hardware architecture to perform the combination of SAL and BC (or BWN) which we name " SALBC block". This architecture uses a simple low-cost multiplexer. In more details, SALBC block is made of two sub-blocks: a memory one and a processing unit one.

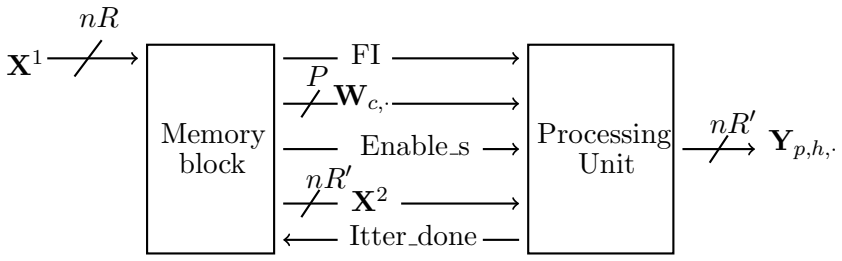


Figure 3.20: Hardware architecture of SALBC block.

The memory block contains two block RAMs (BRAMs) containing data encoded using n bits fixed point. The first is used to store the computed feature maps. Once they are all computed, the content of the first BRAM is copied to the second one, so that it becomes the input of the next layers. At the same time, the computed feature maps of an another image can be stored in the first BRAM. We thus obtain a pipeline architecture, in which all implemented layers work at the same time to speed up inference process.

To avoid data overflow, we process each row of a slice of $\mathbf{X}$ independently, and each slice of the kernel tensor independently. In more details, we copy from BRAM one to BRAM two a feature subvector $\mathbf{X}^2_{c,h} = \{x^2_{c,h,1}, x^2_{c,h,2}, \ldots, x^2_{c,h,R'}\}$ made of R' values, instead of the whole subvector feature vector $\mathbf{X}_{c,h} = \{x_{c,h,1}, x_{c,h,2}, \ldots, x_{c,h,R}\}$ made of $R > R'$ values (*cf.* Figure 3.20). This is to account for the border effects (padding). To simplify notations, we replace $\mathbf{X}_{c,h}$ (resp. $\mathbf{X}^2_{c,h}$) by $\mathbf{X}^1$ (resp. $\mathbf{X}^2$) in the following.

The processing unit uses $\mathbf{X}^2$ and a vector $\mathbf{W}_{c,\cdot}$ made of P values coded on 1 bit each. It thus computes in parallel P feature vectors $\mathbf{Y}_{p,h,\cdot}$ (*cf.* Figure 3.21). The First-Input signal (FI) is set to 1 when the first feature vector is read from the second BRAM to initialise registers by 0. To compute each feature vector p where $1 \leq p \leq P \leq D$, we use the corresponding $w_{c,p}$ to add either $\mathbf{X}^2$ or $\mathbf{-X}^2$ to the content of register p. Once all input feature vectors have been read from the second BRAM of memory block, the signal *Enable_s* is set to 1, and the content of registers is written one by one into the first BRAM of the memory block of the next layer. At the end of this process, the *Itter_done* signal is set to 1 in the processing unit block, so new data can be read from the memory block to process other feature vectors.

To achieve the computation associated with SALBC block described in Figure 3.20, CH clock cycles (CCs) are required to copy all contents from the first BRAM to the second one, CHD/P CCs to compute all output feature vectors of one layer, and DH CCs to write all computed feature vectors into the memory block of the next layer. Thus the total number of CCs required is:

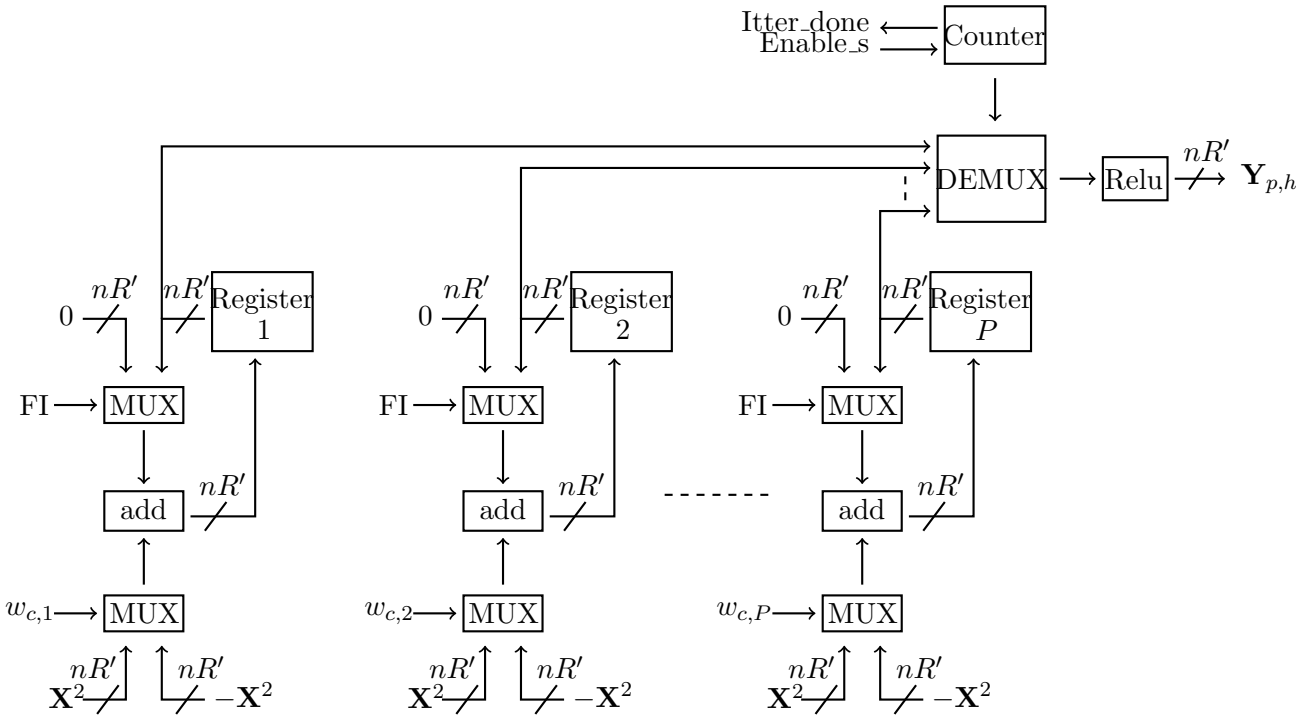


Figure 3.21: Hardware architecture of a processing unit block.

$$CCs = CH + \frac{CHD}{P} + DH. \tag{3.8}$$

This should be compared to [1], where the number of clock cycles becomes:

$$CCs = \frac{3H^2CD}{P}. \tag{3.9}$$

We observe that the proposed architecture is $3H$ faster than the one introduced in [1], which can be significant when H is big. For instance with the CIFAR10 dataset, at the input layer of a CNN $H = 32$, and thus the proposed method is 96 times faster. In addition it is a pipeline architecture, so it can be $3LH$ faster where L is the total number of layers that fit in an FPGA.

Note that in the proposed architecture, P should be lower or equal to D, otherwise reaching full parallelism would require to read more than one vector $\mathbf{X}^2$, and as such would also require more BRAMs, resulting in a more complex architecture.

3.8.2 Hardware Results

We implemented one/few layers of Resnet-18 on Xilinx Ultra Scale Vu13p (xcvu13p-figd2104-1-e) FPGA. The implemented layers are arranged in a pipeline, and their functionality has been verified comparing the output of each SALBC block with the ones obtained by software simulation over a batch of examples. Table 4.5 shows the required resources to implement one/few layers of Resnet-18 trained on CIFAR10 dataset for different values of P. It also shows that the obtained architecture obtain a low processing latency to compute a valid output of one layer. Moreover, this processing latency increases when processing more than one layer, but processing outflow is maintained thanks to the pipeline design.

Table 3.12: FPGA results for the proposed architecture on vu13p (xcvu13p-figd2104-1-e). Here "PL" refers to processing latency.

	P	LUT	FF	BRAMs	Frequency	PL	Processing outflow	Power
Conv64 – 64	16	22424	22424	114	240MHz	52μs	19230 images/s	3.7W
4×Conv64 – 64	16	89746	75235	456	240MHz	208μs	19230 images/s	6.5W
3×Conv128 – 128	32	59780	45024	171	240MHz	154, 8μs	19379 images/s	4.8W
3×Conv128 – 128	64	134090	102552	171	240MHz	103, 2μs	29069 images/s	7.8W
3×Conv256 – 256	64	74067	52051	87	250MHz	147, 3μs	20366 images/s	5.5W
3×Conv256 – 256	128	154599	102723	87	218MHz	112, 8μs	26595 images/s	7.8W
3×Conv512 – 512	128	132155	52151	45	208MHz	177μs	16949 images/s	7.9W

3.9 Energy Gains with Faulty Memories

The large number of parameters and computations makes hardware implementation of DNNs a real challenge that needs a large amount of memory and a complex logic circuit, and thus consumes a significant amount of energy. An easy way to reduce energy consumption is to reduce off-chip memory accesses since they are costly in energy, and use only on-chip memory. However, even when using on-chip memory, the memory access energy represents $30 - 60\%$ of the total energy [47]. One way to reduce energy consumption of both on-chip memory and logic circuit is to reduce the supply voltage. Doing so can cause bit-cell failure and increase failure rates by several orders of magnitude, especially when approaching the minimum energy operating of on-chip memory comparing to operating at the nominal supply [15]. Such a bit-cell failure rate may not be catastrophic if appropriate methods are used to preserve the system's accuracy.

Reducing supply voltage and exploiting fault tolerance to reduce energy consumption has been the main subject of numerous contributions in the last years since DNNs show a limited amount of fault tolerance [100, 45]. For instance, when memory faults are detected at the bit level, a bit masking technique can be used to reduce the magnitude of weights affected by these faults, thus reducing the impact of errors on performance [80, 102]. In [47, 108] the authors propose to modify the training process and take into account bit flips occurring in on-chip memory, and also consider the effect of memory faults when storing the input. In addition, the problem of training a network to compensate known defect locations is considered in [59, 106]

In [27], we investigate the impact of bit-cell faults on DNNs performance, and propose a regularizer to increase the robustness of DNNs when reducing supply voltage. We only consider the energy consumed by memory accesses, and assume that the energy needed to process the inference is proportional to the number of memory accesses. Hence, we denote by E_0 a base energy metric, which represents the sum of the number of all DNNs weights and of the number of activation values used during the inference process.

We consider a model to link bit-cell fault probability p when supply voltage is reduced, and the energy consumed by memory accesses. Let us denote by $0 \leq \eta \leq 1$ the normalized energy consumption in such a way that the energy consumed when reducing supply voltage is given by ηE_0. Considering data published in [15], we could establish a relation between fault probability p and normalized energy η defined as follows:

$$p(\eta) = e^{-a\eta}\,. \tag{3.10}$$

To obtain a specific value of a, we consider the energy data reported in [8] and the reliability of on-chip memory for 65nm CMOS at $V_{\mathrm{DD}} \in \{0.5, 1.1\}$ from [15]. Minimising the sum of the relative squared error leads to $a = 12.8$. In our study we consider the case when bit-cell faults can be detected, and then used the bit masking (BM) deviation approach introduced in [80]. The BM approach can be defined as follows: when a memory fault is detected on the sign bit, the corresponding value is then replaced by zero. On the other hand, when a memory fault is detected on any other bit, the bit value is replaced by sign bit value. We consider that all bit cells have an equal memory fault probability p, and memory faults can affect both weights and activations. Note that due to the use of the activation function ReLU, activation values are positive, and then we assume that memory faults cannot affect their sign bit.

To study the robustness to memory faults, we perform experiments using CIFAR10, and compare four main architectures, PreActResNet18 [31], MobileNetV2 [86],

Table 3.13: Number of memory accesses and accuracy by architecture

Architecture	Parameters	Activations	Accuracy
PreActResNet18 [31]	11.2×10^6	0.55×10^6	94.87%
MobileNetV2 [86]	2.30×10^6	1.53×10^6	93.80%
SENet18 [38]	11.3×10^6	0.86×10^6	94.77%
ResNet18 [30]	11.2×10^6	0.56×10^6	94.86%

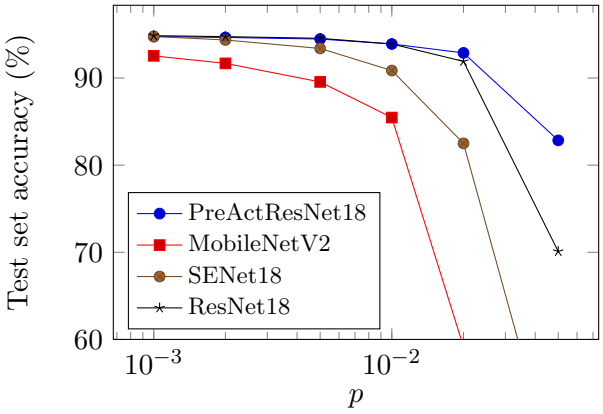


Figure 3.22: Impact of the architecture on the robustness under BM deviations.

SENet18 [38] and ResNet18 [30], which represent modern CNNs architectures achieving a good accuracy on CIFAR10. Table 3.13 shows the obtained accuracy and the number of weights and activations needed to process one input image for each CNN architecture.

We perform a first experiment in which we compare the robustness of the different CNN architectures mentioned above when both weights and activations are affected by BM. Figure 3.22 shows the accuracy behaviour when varying the memory fault probability p, and Figure 3.23 plots the accuracy in function of energy ηE_0, where E_0 represents the sum of weights and activations reported in Table 3.13, and p is obtained from the normalized energy η as described in Equation (3.10). From both Figures 3.22 and 3.23, we see that some architectures are more robust than others, and PreActResNet18 provides a good trade-off between accuracy, number of parameters and activations and robustness to BM, thus we focus on this architecture when performing other experiments.

In a second experiment, we want to identify the relative robustness of different parts of the CNN when applying BM deviations. To do this, and since PreActResnet18 is made of 4 sequential blocks (each one contains 2 convolutional layers, 2 batch-norm layers and 1 shortcut), we apply BM deviation to both weights and activations of one block at a time. Figure 3.24 plots the obtained results, and shows that all neural network blocks

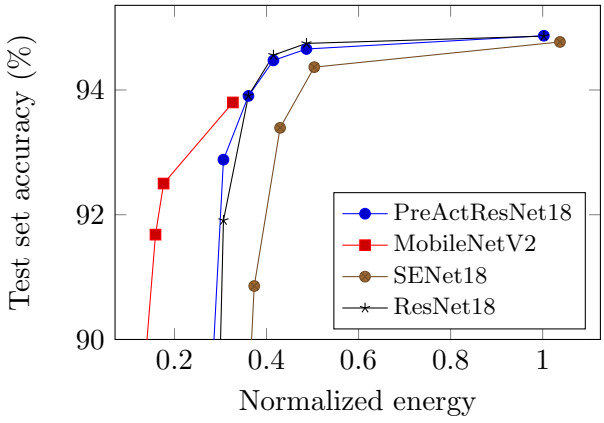


Figure 3.23: Energy consumption of different architectures under BM deviations.

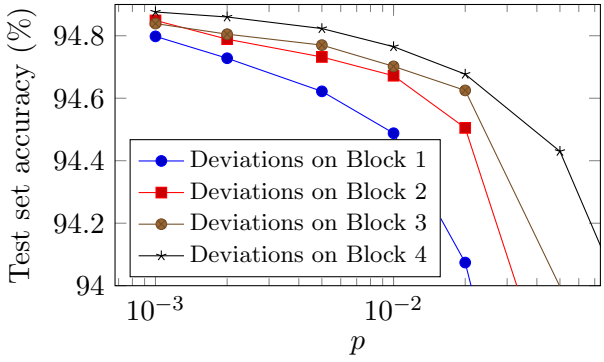


Figure 3.24: Impact on accuracy of BM deviations applied to different stages of the network, "Block 1" being the first and "Block 4" the last.

are affected by BM deviations. Moreover, we observe that the robustness is increased with the depth of the neural network. According to this, a clever method to reduce energy consumption while keeping a good accuracy will be to exploit this difference of robustness through different layers. We will denote this approach by "Diff Fault" in the remaining experiments. We also notice that at a high accuracy of 94.8%, we have $p_{\mathrm{B4}} = 5p_{\mathrm{B3}} = 5p_{\mathrm{B2}} = 10p_{\mathrm{B1}}$, where p_{Bi} is the memory fault probability assigned to block i. This configuration is used in the following when Diff Fault is introduced.

Another way to reduce energy consumption and keep a good accuracy is to apply the deviation model during training in order to increase the robustness of DNNs. Since training is computationally expensive, and since the BM deviation model deviates values towards zero, we propose to replace it by a less complex deviation model in which each value has a probability p_e to be zero, referred to as the erasure model. To do so, we need to find a way to link memory faults probability p and the probability for a value to be zero p_e. During training, erasure model is similar to dropout [90], but in this case it is used to increase DNNs robustness rather than to prevent overfitting. To find a good

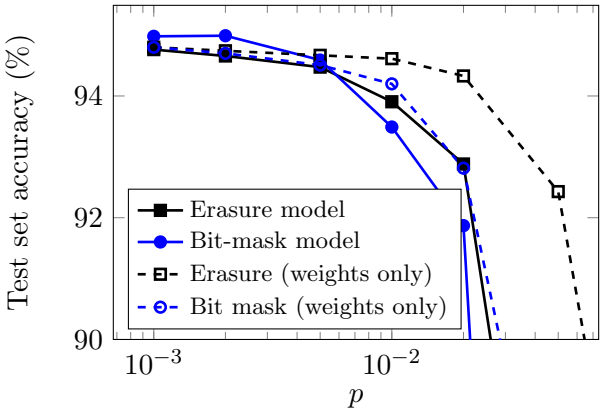


Figure 3.25: Impact of memory faults on accuracy for different deviation models.

function f such as $p_e = f(p)$, we evaluate the effect of both BM and erasure models on PreActResNet18 and we plot results in Figure 3.25. Because the number of weights in PreActResnet18 is 20× more than activations, we only consider when deviation models are applied on weights only, and match the accuracy of the two models to find f. From Figure 3.25, we observe that BM and erasure model reach the same accuracy when $p_e = 2p$, and thus this relation allows to use erasure model as an approximation of BM. Using erasure model during training is referred to as regularizer (reg).

We also consider the effect of reducing the number of parameters on the accuracy. Since the number of parameters (weights) linearly depends on both the number of input and output feature maps, an easy way to reduce it will be to reduce the number of feature maps, such as if the number of feature maps is divided by a number k, the number of parameters will be divided by k^2. As a reference we train two variants of PreActResNet with $F/2$ and $F/\sqrt{2}$, where F represents the original number of feature maps.

As a last experiment, we aim at providing the effect of deviations when applying erasure model during training. Note that we use erasure model rather than BM because it is less complex and then speed up the training process. Figure 3.26 shows that introducing erasure model during training allows to achieve same accuracy as standard training while using less energy. Moreover, we notice that combining erasure regularizer with Diff Fault leads to an additional gains. We thus conclude that we can significantly improve the energy reduction using erasure regularizer and Diff Fault during training. In addition, an interesting thing we notice is that to reduce energy consumption, it is better to train a bigger neural network for robustness than just reduce the neural network size.

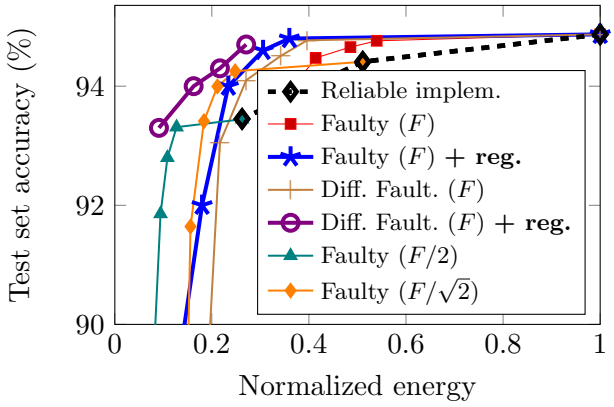


Figure 3.26: Energy consumption of the Preact-Resnet18 architecture under BM deviations. Each faulty implementation curve corresponds to a fixed network size, with the number of feature maps shown within parentheses.

3.10 Summary of the Chapter

We discussed in this chapter quantization techniques used to compress DNNs size and reduce their computations and complexity. We proposed Shift Attention Layer (SAL), a shift module based method, and a guided pruning method that aims at replacing standard convolutional by a shift operation followed by 1×1 convolution. We saw that such a method eases hardware implementation of DNN based solution on FPGA.

We also proposed to study the effect of input power voltage of an embedded system on DNNs robustness, and proposed a regularizer to make DNNs more robust against voltage drop. Finally, we proposed to compare some quantization techniques to see which method achieves the best trade off between accuracy and memory footprint.

A logical continuation to this work would be focusing on how to reduce complexity of the training process of SAL, and continue exploring quantization methods and their combination to find the architecture that achieves the best accuracy for a given energy budget. Another challenge would be to avoid storing the whole dataset needed during learning phase, and perform incremental learning where only one or few examples are stored and learn at a time.

Chapter 4

Incremental Learning on Chip

Contents

4.1 Context

During a life time, humans have the ability to learn incrementally new pieces of information, combining them to previously acquired knowledge when facing day-to-day tasks. This process is nondestructive, and usually called in the literature "curriculum learning" [3]. By contrast, Deep Neural Networks (DNNs), although they were introduced as a simplifying model for brain mechanism, cannot achieve the same kind of learning. Indeed, training with streaming data has the consequence of destroying previously learned

knowledge and results in what is usually referred to as "catastrophic forgetting" in the literature [46, 17].

Despite the fact that DNNs became the state-of-the-art in several domains, they are still unable to perform an incremental learning process because learning new data will modify DNNs parameters in such a way to lose previously acquired information. Many techniques try to avoid this loss of knowledge by learning several DNNs over time, and use another algorithm to choose which DNN is more adapted to process an input data [19, 72]. Such a technique leads to very complex systems quickly, and are likely to fail in adversarial conditions [97].

Let us first define what incremental learning refers to (adapted from [81]):

1. It is able to perform learning process using one or few examples at a time, without requiring to store or consider previous learned examples.
2. It is able to approach state-of-the-art classification accuracy while learning incrementally new data, and thus avoid catastrophic forgetting.
3. It requires low computational power and memory footprint during both learning and inference phases.

Incremental learning has received a particular interest for a long time [87, 99, 119], and several methods have been proposed. However, satisfying criteria listed above while keeping a high accuracy remains an open challenge.

There is no doubt that DNNs are state-of-the-art in many machine learning challenges. But they rely on large quantities of available data and hundreds of millions of trainable parameters to perform the learning process, which require a large memory footprint and computational power, and thus makes Learning On Chip (LOC) an open challenging research so far [6, 74, 71, 53]. Due to the complexity of DNNs and the resources needed to perform a learning phase, most recent works propose DNN hardware implementations targeting only the inference part [5, 58, 109, 23], and assume that the learning phase is computed offline using a remote server. Incremental learning approaches satisfying the above-mentioned criteria would be a good solution to overcome LOC problems, since they learn only one or few examples at a time, and do not use a large memory to store data. However, the methods presented in the literature often achieve poor accuracy compared to DNN counterparts. In the coming sections, we explain how to combine incremental approaches with DNNs to achieve high accuracy while performing Incremental Learning On Chip (ILOC). The chapter is organised as

follows. In Section 4.2 we present incremental learning related works. In Section 4.3 we introduce transfer learning concept. In Section 4.4 we discuss vector segmentation and how it helps to classify feature vectors obtained using transfer learning. Section 4.5 and Section 4.6 explain two incremental learning methods of our contribution. Section 4.7 performs some experiments using challenging computer vision datasets. In Section 4.8 we propose a hardware architecture and show some FPGA implementation results of an ILOC solution, and finally in Section 4.9 we summarise the chapter.

4.2 Main Methods in the Literature

There has been some interest in incremental learning during last years [61]. For instance, in [93, 76, 117], the authors propose a Support Vector Machine (SVM) based method to learn one subset at a time. To learn a batch of new data, a new SVM is trained on these new data combined to support vectors of previous SVMs. Since support vectors are not conveying the full extent of previous data, the new resulting SVM will suffer from catastrophic forgetting, and thus does not fulfill criterion 2 introduced in Section 4.1.

"Learn++" [78, 69], another incremental learning algorithm, uses weak one-vs-all classifiers to accommodate new classes and combine them through weighted majority votes. This approach is also able to manage the insertion, deletion and recurrence of classes over learning data [92]. However, it needs to add and train new classifiers each time a new class is introduced, and then ends up with a large computational power and memory footprint which violates criterion 3. This method is also used to add the incremental learning capability to SVMs, by using a set of SVMs trained with Learn++ called "SVMLearn++" [16], which consists of using the learn++ algorithm with an SVM classifier. Despite the fact that SVMLearn++ shows promising results on biological datasets [68], this method still needs to train new SVMs each time new data is available, and suffers from catastrophic forgetting.

Pentina *et al.* [75] show the possibility of learning data sequentially. However, to perform such an operation, they need to choose a correct ordering of the whole dataset, and thus have the whole dataset before training which violates criterion 1. In [66], the authors propose a transfer learning technique (*cf.* Section 4.3), in which a pre-trained DNN is used as feature extractor, followed by the Nearest Class Mean classifier (NCM). NCM summarizes each class using the average feature vector $\bar{\mathbf{X}}$ of all examples observed for the class so far. To classify a D-dimensional feature vector $\mathbf{X}$ given by the pre-trained DNN, NCM assigns to it the class of the closest mean as follows:

$$y^* = \underset{y \in \{1,\dots,Y\}}{\operatorname{argmin}} \; d(\mathbf{X}, \bar{\mathbf{X}}) \tag{4.1}$$

$$\bar{\mathbf{X}} = \frac{1}{N_y} \sum_{i:y_i=y} \mathbf{X}^i \,, \tag{4.2}$$

where $d(\mathbf{X}, \bar{\mathbf{X}})$ is the Euclidean distance between $\mathbf{X}$ and $\bar{\mathbf{X}}$, y_i is the label of $\mathbf{X}^i$, and N_y is the number of samples in class y. The authors also propose to use learnable metric instead of Euclidean distance during classification. However, to do so, they need to use the whole dataset to learn the new metric, which does not correspond to an incremental learning concept. NCM shows a better accuracy in incremental learning scenario compared to other parametric classifiers [65, 66, 82], but lower than state-of-art, and hence does not fulfill criterion 2.

In [52], Kuzborskij *et al.* show the possibility of adding new classes to a multi-class classifier while keeping an acceptable accuracy. The classifier can be retrained using a small amount of data belonging to all classes. Based on this work, in [81] the authors propose "Incremental Classifier and Representation Learning" (iCaRL), an incremental learning method using a trainable DNN feature extractor, and an NCM classifier. The classification process is the same as introduced by the NCM method, where a DNN is used as a feature extractor, and the class of the nearest mean is assigned to the obtained D-dimensional feature vector $\mathbf{X}$. During the learning process, the authors use a loss function containing a classification term that encourage the network to output the corresponding class of a new image, and a distillation term which ensures that previously learned information is not lost when new classes are learned. Note that m images per each learned class are kept, and combined to new input data to retrain the model, which violates criterion 3. Moreover, when given a data stream containing only few classes at a time, iCaRL achieves a very low accuracy as depicted in [81, 7], hence iCaRL does not fulfill criterion 2. On the contrary, to reach good performances and a comparable accuracy to state-of-art methods, iCaRL thus needs to be trained over batches of data containing a large part of the dataset, which does not correspond to an incremental learning scenario and infringes criterion 1.

We introduce Budget Restricted Incremental Learning (BRIL) [24], and Transfer Incremental Learning using Data Augmentation (TILDA) [7], two incremental learning methods using a pre-trained DNN as feature extractor, and an incremental classifier trained on obtained feature vectors. In these methods, we propose to improve the accuracy of transfer learning using vector segmentation.

4.3 Transfer Learning

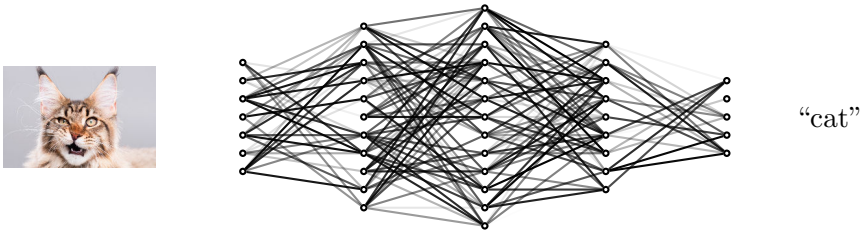
During the past few years, transfer learning based on DNNs as feature extractors has become increasingly popular [110]. It is used to reach state-of-the-art accuracy on too small datasets that cannot be used to train a neural network, or to avoid the large computational training of DNNs. Basically, transfer learning consists first in training a deep neural network on a large first dataset, and then using inner layers of the obtained pre-trained DNN that act as a generic feature extractor [70, 35, 72], combined with classification methods such as Multi Layer Perceptron (MLP), Support Vector Machines (SVMs) or Nearest Neighbour search (NN) to process a second dataset (*cf.* Figure 4.1). As a matter of fact, DNN's inner layers provide a good description of an input image, even when it does not belongs to the learning domain [70]. A transfer learning based method allows a rapid, flexible and low cost deployment of performing solutions in restricted embedded systems such as robots or smartphones, since the larger and computational part consists of a pre-trained unchanged DNN. In the next section, we discuss feature vectors obtained from DNN's inner layers and how the classification accuracy can be improved when using vector segmentation.

4.4 Segmentation

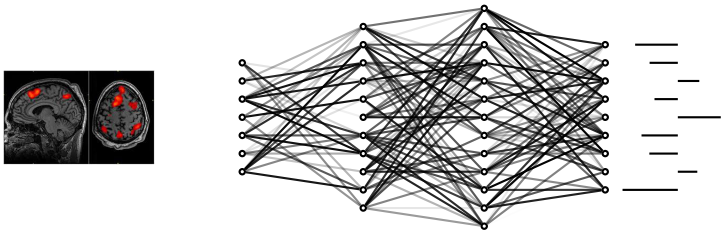
Segmentation is the process of partitioning and splitting a given vector into subvectors, process each subvector independently to obtain a result and compute an algorithm that combines all obtained results for all subvectors such as a majority vote to finally get a result corresponding to the initial vector.

The reason why vector segmentation helps to increase the accuracy is directly linked to the use of transfer learning. Indeed, to provide a good representation of feature vectors, the pre-trained DNNs that are used to compute transfer learning were trained on a dataset containing a large variety of classes. As a consequence, it is expected that a considerable part of the extracted feature vectors of a dataset counting few classes is not used. Hence, the useful information in the resulting feature vectors is likely to be sparsely spread among the coordinates. In [22], we show that for some distributions where feature vectors are sparse and information is represented by only few coordinates, splitting D-dimensional feature vectors into P equal size parts, where $1 < P << D$, classifying each part independently and then performing a majority vote to classify the feature vector can help a non-parametric classifier (e.g. nearest neighbour search (NN)) achieve a better accuracy than just classifying the feature vector.

1. Train a CNN using massive *generic* datasets:



2. Compute feature vectors using an intermediate representation in the CNN:



3. Use a classification method on obtained feature vectors:

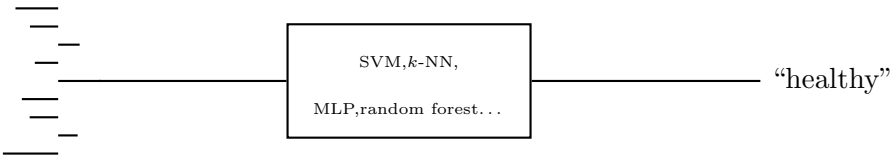


Figure 4.1: Overview of transfer learning process.

Inception V3, 1-NN

P	1	4	16	64	256
CIFAR10	0.8519	0.8652	**0.8781**	0.8651	0.8347
ImageNet1	0.9328	0.9354	0.9424	**0.9439**	0.9081
ImageNet2	0.9438	0.9451	**0.9524**	0.9464	0.9171

Inception V3, 5-NN

P	1	4	16	64	256
CIFAR10	0.8689	**0.8761**	0.8759	0.8668	0.8461
ImageNet1	0.9389	**0.9450**	0.9429	0.9394	0.9202
ImageNet2	0.9467	0.9498	**0.9511**	0.9488	0.9303

SqueezeNet, 1-NN

P	1	5	20	100	200
CIFAR10	0.6839	0.7069	**0.7472**	0.6890	0.6225
ImageNet1	0.8854	0.8900	**0.9001**	0.8784	0.8466
ImageNet2	0.8737	0.8802	**0.8926**	0.8669	0.8267

SqueezeNet, 5-NN

P	1	5	20	100	200
CIFAR10	0.7284	0.7483	**0.7566**	0.6954	0.6371
ImageNet1	0.8985	0.8965	**0.8980**	0.8698	0.8501
ImageNet2	0.8862	**0.8901**	0.8893	0.8591	0.8280

AudioSet

P	1	2	10	20	40	160
1-NN	0.605	0.621	0.704	0.698	**0.724**	0.660
5-NN	0.564	0.649	0.704	0.718	**0.727**	0.668

Table 4.1: Accuracy of classification, depending on the feature extractor used, the dataset and the number of segments P. This table is introduced in [22].

To show the effect of splitting feature vectors on accuracy, we use three main pre-trained DNNs as feature extractors: InceptionV3 [96] trained on ImageNet ILSVRC 2012 that outputs a 2048-dimensional feature vector, SequeezeNet [40] trained on ImageNet ILSVRC 2012 as well that outputs a 1000-dimensional feature vector, and a DNN trained on AudioSet [18] that outputs a 1280-dimensional feature vector which represents the concatenation of ten 128-dimensions feature vectors, one per second of the corresponding audio track. We perform our tests on CIFAR10, Imagenet1, ImageNet2, and on 10 classes chosen in AudioSet so that they contain a similar number of elements (radio, cat, hi-hat, helicopter, fireworks, stream, bark, baby/infant cry, snoring, train horn). We report the results in Table 4.1. Table 4.1 shows that for each experiment, splitting feature vectors into P parts leads to a better accuracy. We exploit this idea when introducing BRIL in Section 4.5 and TILDA in Section 4.6 to improve classification accuracy.

4.5 Budget Restricted Incremental Learning

We introduce in [24] Budget Restricted Incremental Learning (BRIL), an incremental learning method built upon three main steps: 1) the use of a pre-trained DNN to extract feature vectors from an input dataset, 2) the use of product quantization techniques to embed data in a finite alphabet and 3) the use of a majority vote to classify data.

The first step consists in using transfer learning to extract features of a given input. Indeed, inner layers of a DNN pre-trained on a large number of examples act as a generic feature extractor. In the following, we denote by $\mathbf{X}_0^m$ the m-th training input and by $\mathbf{X}_l^m$ its corresponding feature vector, where $1 \leq m \leq M$, and M is the total number of training inputs (*cf.* Figure 4.2 step 1).

The second step consists in quantizing obtained feature vectors $\mathbf{X}_l^m$ where $1 \leq m \leq M$ using a Product Quantization (PQ) technique [42]. Since we aim at providing a computationally light solution (*cf.* criterion 3), we choose to use Product Random Sampling as a PQ technique. Basically, we split each feature vector $\mathbf{X}_l^m$ into P disjoint sub-vectors of equal size $\left(\mathbf{X}_{l,p}^m\right)_{1\leq p\leq P}$, and quantize each resulting sub-vector independently from each other using k randomly sampled anchor vectors $(\mathbf{V}_{p,i})_{1\leq p\leq P, 1\leq i\leq k}$, where each $\mathbf{V}_{p,i}$ is such that $\exists \mathbf{X}_l^m, \mathbf{X}_{l,p}^m = \mathbf{V}_{p,i}$. In the remaining of this chapter, we refer to $(\mathbf{V}_{p,i})_{1\leq p\leq P, 1\leq i\leq k}$ as anchor sub-vectors.

Next, each sub-vector $\mathbf{X}_{l,p}^m$ is quantized by choosing the closest anchor sub-vector in its corresponding subspace, as depicted in Equation (4.3). We use the Euclidean distance to determinate the closest anchor sub-vector. Each quantization is independent from each other, so that the process can be performed concurrently, enabling a highly parallel implementation on hardware. Note that since we are using Product Random Sampling, the learning phase only consists in computing feature vectors of input data, splitting these feature vectors into P equal size parts, and then choosing randomly k sub-feature vectors to store, representing anchor vectors and their corresponding classes. Learning new data results only into storing new anchor vectors. The parameter k controls how many anchor vectors are added to each class, and thus most of training data is disregarded when k is a small number.

$$\begin{cases} i^\star(m,p) &= \arg\min_i \|\mathbf{X}_{l,p}^m - \mathbf{V}_{p,i}\|_2 \\ \mathbf{Q}_p^m &= \mathbf{V}_{p,i^\star(m,p)}. \end{cases} \tag{4.3}$$

Finally, as a last step, we identify the classes that correspond to the obtained $(\mathbf{Q}_p)_{1\leq p\leq P}$, and perform a majority vote using theses classes to take a final decision and

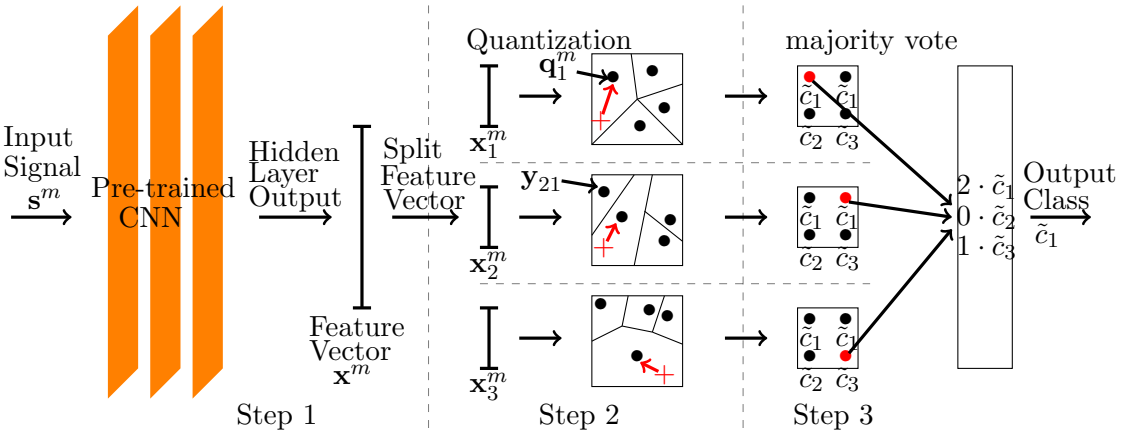


Figure 4.2: Overview of the proposed method, comprising three main steps. Given a set of samples, we first use a pre-trained CNN for feature extraction (Step 1). Subsequently, we use a PQ technique to quantize the feature vectors (Step 2). Finally, we use a majority vote to classify the quantized data (Step 3). This figure is introduced in [24].

classify the input $\mathbf{X}_0^m$. The combination of using a pre-trained DNN as feature extractor and a majority vote classifier allows the model to learn new classes and/or examples without damaging previously learned knowledge [21] or retraining it. BRIL constitutes our first original proposal for an incremental learning method. However, as we will see in the following benchmarks, and despite being compliant with criteria 1 and 3, BRIL violates criterion 2 since it achieves a significantly lower classification accuracy than state-of-the-art methods. In the next section, we introduce another method, TILDA, before moving on to benchmarking the two proposed approaches.

4.6 Transfer Incremental Learning using Data Augmentation

In [7], we introduce Transfer Incremental Learning using Data Augmentation (TILDA), an incremental learning method that attempts to combine the characteristics of previously introduced work to fulfill all 3 criteria.

In more details, TILDA uses a pre-trained DNN to extract features from input vectors, as with the iCaRL and BRIL methods. As BRIL, TILDA uses vector segmentation to improve the accuracy. TILDA uses NCM-based classifiers to reduce the memory footprint. Finally, as BRIL and Learn++, TILDA uses a majority vote to aggregate the decisions of multiple classifiers.

TILDA process can be split into four main steps: 1) a pre-trained DNN extracts feature vectors, 2) feature vectors are split into multiple subvectors, 3) each subvector is classified independently from the others using an NCM-based method, and 4) the multiple decisions are aggregated using a majority vote.

Note that in order to further increase the accuracy of the method, we not only use data augmentation during training but also when predicting the class of a given unlabelled input. In other words, we generate multiple versions of a unlabelled input, obtain a decision for each one then perform a majority vote (distinct from the one of step 4) to obtain a global decision.

In the coming subsections, we review in detail each of the above mentioned steps.

4.6.1 Feature Vector Extraction

Similarly as in BRIL, to perform feature extraction, TILDA relies on the use of a pre-trained DNN. We will use the same notation as defined above, for which the feature extraction leads to consider the feature vector $\mathbf{X}_l^m$ during learning and classification process instead of its corresponding input $\mathbf{X}_0^m$, where l denotes one layer in the DNN architecture. Since we consider a fixed layer l, for more readability we denote $\mathbf{X}_y^m \triangleq \mathbf{X}_l^m$, where y is the class of $\mathbf{X}_l^m$.

4.6.2 Vector Segmentation

As discussed in Section 4.4, we split each feature vector $\mathbf{X}_y^m$ into P equal size parts, denoted $\left(\mathbf{X}_{y,p}^m\right)_{1\leq p\leq P}$. For each class and subspace, we use k anchor vectors initialized as 0. We associate to each anchor vector a counter, also initialized by 0, which represents how many times the corresponding anchor vector has been modified. Considering each subspace p and each class y, we denote by $\mathbf{V}_{y,p} = [\mathbf{V}_{y,p,1}, ..., \mathbf{V}_{y,p,k}]$ the corresponding anchor vectors and $N_{y,p} = [N_{y,p,1}, \ldots, N_{y,p,k}]$ their associated counters.

For each class y and subspace p, anchor vectors should be interpreted as centroids of a clustering of the corresponding subspace with observations $\{\mathbf{X}_{y,p}^m\}$. In other words, at each step of the training process we ensure that anchor vectors are the barycenter of a subset of already processed input sub-vectors, and the associated counter accounts for the cardinality of the corresponding subset.

Then, each time an input training vector is processed, we identify which anchor vector we need to update. The update process consists of computing a new anchor

vector which represents a barycenter of the old one with weight given by its counter and the input training sub-vector with weight 1, and then incrementing the corresponding counter by one. Basically, this is an online way to compute the average of the subset of vectors associated with a given anchor vector.

A problem with clustering methods when they are performed in an online manner is that they are likely to cause unbalanced clusters. In order to avoid this, we penalize most used anchor vectors by taking into account their corresponding counters when associating a new input subvector to its corresponding cluster. More precisely, for each class y and subspace p, we multiply obtained distances $(d_i)_{1\leq i\leq k}$ between input training sub-vector $\mathbf{X}_{y,p}^m$ and anchor vectors $\mathbf{V}_{y,p}$ by corresponding counters $N_{y,p}$, and then associate $\mathbf{X}_{y,p}^m$ to $\mathbf{V}_{y,p,i}$ corresponding to the smallest $d_i N_{y,p,i}$. This procedure is detailed in Algorithm 2. Note that when two or more anchor vectors obtain the same score (*i.e.* distances multiplied by counters), we choose uniformly at random one of the them.

Algorithm 2 Incremental Learning of Anchor Subvectors

Input: streaming feature vector $\mathbf{X}_{y,p}^m$

for $p := 1$ to P **do**
  for $i := 1$ to k **do**
    $d_i = \|\mathbf{X}_{y,p} - \mathbf{V}_{y,p,i}\|_2$
    $R_i = d_i n_{y,p,i}$
  end for
  $\tilde{k} = \arg\min_i R_i$
  $\mathbf{V}_{y,p,\tilde{k}} \leftarrow \mathbf{V}_{y,p,\tilde{k}} n_{y,p,\tilde{k}} + \mathbf{X}_{y,p}^m$
  $n_{y,p,\tilde{k}} \leftarrow n_{y,p,\tilde{k}} + 1$
  $\mathbf{V}_{y,p,\tilde{k}} \leftarrow \mathbf{V}_{y,p,\tilde{k}} / n_{y,p,\tilde{k}}$
end for

Note that the way we perform the clustering is unfortunately not independent of the order of the streaming data, which contradicts criterion 1. However, it is possible, at the cost of a lower accuracy, to change the clustering technique to fulfill this criterion.

4.6.3 Aggregation of Subspaces Weak Classifiers

At prediction stage, given an unlabelled input $\mathbf{X}_0$, we first compute its corresponding feature vector $\mathbf{X}$ using the pre-trained DNN. We then split $\mathbf{X}$ into P parts and obtain

$(\mathbf{X}_p)_{1\leq p\leq P}$. We compute Euclidean distances between each $\mathbf{X}_p$ and all anchor vectors $\mathbf{V}_{y,p,i}$ for which the counter is not 0. Note that there are at most kY such distances, where Y is the number of classes seen so far. The class of the closest average anchor vector is considered as the decision for the p-th subspace. Finally, we apply a majority vote over all subspaces to achieve an aggregate decision (*cf.* Algorithm 3). Note that more elaborate strategies such as a weighted majority vote can result in higher accuracy but may require more computation during the learning phase as well as memorization of previously seen examples.

Algorithm 3 Predicting the Class of a Test Input Signal

Input: input signal **s**

Compute the feature vector **X** associated with **S**

Initialize the vote vector **C** as the **0 vector** with dimension Y

for $p := 1$ to P **do**

$v_p = \arg\min_y \left[\min_i \|\mathbf{X}_p - \mathbf{V}_{y,p,i}\|_2\right]$

$\mathbf{C}_{v_p} = \mathbf{C}_{v_p} + 1$

end for

$\tilde{y} = \arg\max_y(\mathbf{C}_y)$

Output: class $\tilde{y}$ attributed to **s**

4.6.4 Data Augmentation

We use two data augmentation methods to improve the accuracy and robustness: one during training and one during classification.

During Training

To improve the accuracy without increasing memory usage, data augmentation is applied to the training dataset. We generate multiple versions of each training input (*cf.* Section 4.7.1), and consider the resulting dataset as an input to train the model.

During Classification

In addition, we propose to obtain multiple predictions for each unlabelled input $\mathbf{X}_0$ using data augmentation [10]. The idea is to generate multiple versions of the input $\mathbf{X}_0$ that we denote $(\mathbf{X}_{0,s})_{1\leq s\leq S}$. We perform a prediction of the class associated with each $\mathbf{X}_0$ independently, and then perform a second a majority vote to obtain the final prediction.

Remarks

We point out multiple facts about the proposed method:

1. The learning procedure performs learning one example at a time,
2. The learning procedure is computationally light as it only requires performing of the order of D operations where D is the dimension of feature vectors,
3. The learning procedure has a small memory footprint, since it only stores the averages of feature vectors,
4. The learning procedure is such that adding new examples can only increase robustness of the method, so that there is no catastrophic forgetting,
5. During prediction stage, memory usage is of the order of kYD and thus is independent on the number of examples and grows linearly with the number of classes,
6. During prediction, computations are of the order of $kYDS$ elementary operations.

From these facts we derive that TILDA is compliant with criteria 1 and 3 defined in the introduction. In the next section, we devise a set of experiments to evaluate the classification accuracy of the proposed method on challenging datasets (criterion 2).

4.7 Experimental Results

To evaluate and compare some incremental and non-incremental learning methods, we use a benchmark protocol described in the following section.

4.7.1 Benchmark Protocol

We propose an incremental learning scenario in which the streaming data may contain new classes and/or new examples. We test and compare Budget Restricted Incremental Learning (BRIL), Nearest Neighbour search (NN), the Nearest Class Mean classifier (NCM), Learn++, incremental Classifier and Representation Learning (iCaRL), and Transfer Incremental Learning using Data Augmentation (TILDA). Note that Learn++ uses Classification And Regression Trees (CART) as weak classifiers.

We perform the evaluation on some challenging computer vision datasets: CIFAR10, CIFAR100 and ImageNet ILSVRC 2012 [85]. Because all methods use a DNN pre-trained

on ImageNet ILSVRC 2012, we also use 50 classes extracted from the wider ImageNet dataset that have not been used to train the CNN (denoted ImageNet50). All methods take the same feature vectors extracted from Inception V3 [96] as input and use the whole dataset for training, unless explicitly mentioned. This requires to modify iCaRL method by replacing its CNN with a fully connected network: we use a MultiLayer Perceptron (MLP) with one hidden layer containing 1024 neurons, and output layer containing Y neurons, where Y is the number of classes.

Note that CIFAR10, CIFAR100 and ImageNet could be arguably considered as similar datasets, since they all contain pictures of various common objects. As such, we expect to reach high accuracy when using a pre-trained DNN on ImageNet to predict the classes for CIFAR10 or CIFAR100.

We also compare TILDA with non-incremental learning methods (NI) denoted by TMLP and TSVM. TMLP uses transfer learning to compute feature extractors through Inception V3, and then trains an MLP over feature vectors, using the hyper-parameters previously described for iCaRL. TSVM method uses Inception V3 to get feature vectors as well, and uses them to train an SVM using Radial Basis Function kernel.

Data augmentation used in TILDA generates a horizontal flip of the original image, and shifts the pixels of the image by one pixel at a time (to the left, right, top, bottom, and on the four diagonals). Thus we generate $S = 10$ images (8 generated by shifting pixels on the image, one generated by horizontal flip and the original one).

4.7.2 Results

In the first experiment, we consider only the proposed TILDA method, and we aim to show that replacing the last layers of Inception V3 by TILDA does not compromise the performance obtained on Imagenet ILSVRC 2012. The 5-top accuracy is 94.4% when we use TILDA with $P = 16$ and $k = 30$, and 96.5% when we use the last layers of Inception V3 to classify data. So the accuracy obtained when using TILDA approaches the one obtained by the full pre-trained Inception V3.

In the second experiment, we consider only TILDA as well, and we depict the contribution of each TILDA's step (*i.e.* vector segmentation, NCM-inspired classification and data augmentation) on classification accuracy. This kind of experiments is often referred to as an ablation test in the litterature.

Therefore, we define three methods: TILDA-DA does not use data augmentation

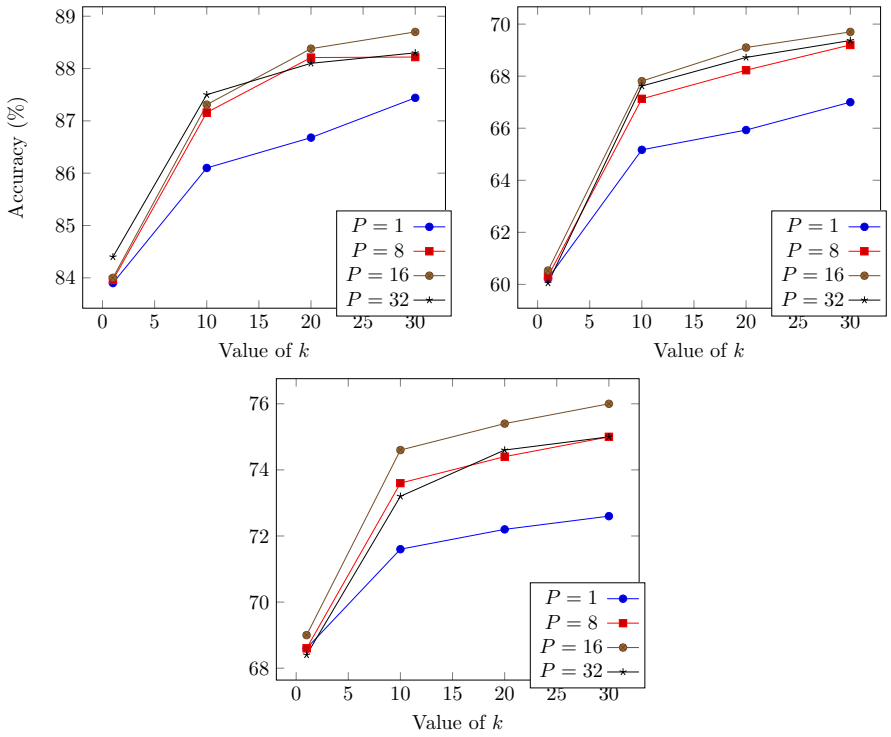


Figure 4.3: Evolution of the accuracy as a function of P and k for CIFAR10 (left), CIFAR100 (right) and ImageNet50 (bottom). This figure is introduced in [7].

and classifies only the original image, TILDA-NCM disregards NCM inspired classification and uses k feature vectors randomly chosen per class, and TILDA-P where vectors are not splitted. The evaluation is performed on CIFAR10, CIFAR100, ImageNet50 and ImageNet ILSVRC 2012. Table 4.2 summarizes the accuracies of TILDA, TILDA-DA, TILDA-NCM and TILDA-P, when performing one-shot learning (learning one example at a time). We notice that TILDA-DA, TILDA-NCM and TILDA-P reach lower accuracies than TILDA, which confirms the interest of the combination of data augmentation with NCM-inspired classification and subspace division.

One more time, we perform an experiment in which we consider only TILDA to study the effect of both quantization parameters P and k on the accuracy (*cf.* Figure 4.3). This experiment demonstrates that TILDA reaches best performances for $P = 16$. In the following, we perform experiments using TILDA with $P = 16$ and $k = 30$. Note that in order to be fair in comparison with other techniques, we do not perform data-augmentation during training or prediction in TILDA in the upcoming experiments.

As a fourth experiment, we aim at stressing the effect of class-incremental learning. We adopt a class-incremental scenario (CI), in which methods are trained over streaming data providing all examples from one class simultaneously, one class at a time. We test

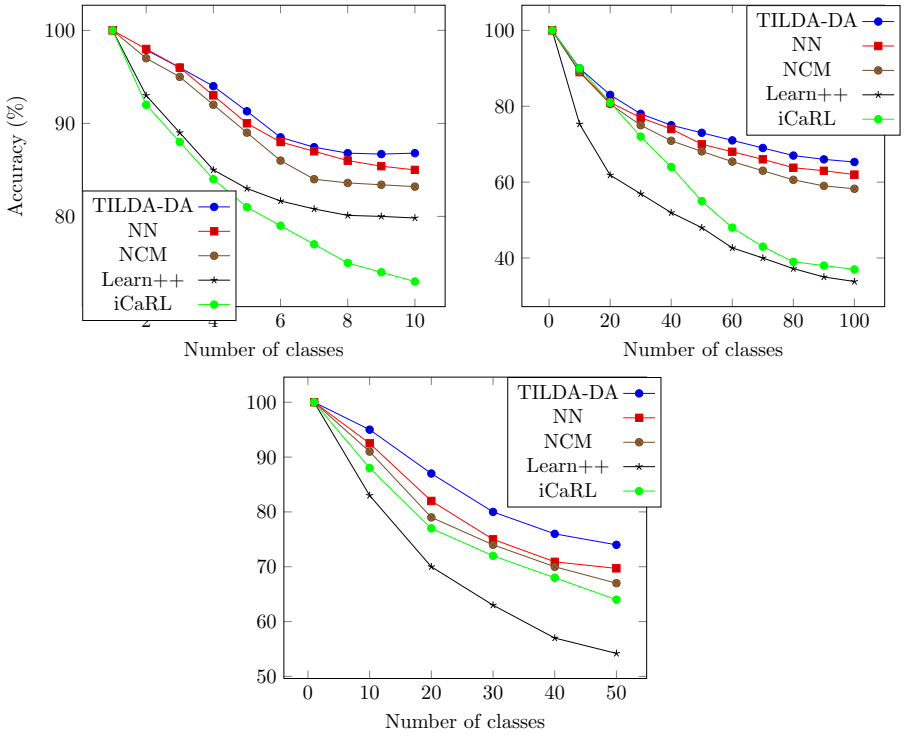


Figure 4.4: Evolution of the accuracy as a function of number of classes for CIFAR10 (left), CIFAR100 (right) and ImageNet50 (bottom). This figure is introduced in [7].

and compare TILDA-DA, NCM, Learn++, NN and iCaRL on CIFAR10, CIFAR100 and ImageNet50 (*cf.* Figure 4.4). Learn++ adds one weak classifier each time a novel class is introduced, and iCaRL stores 30 feature vectors per class. We can see that TILDA-DA outperforms the other methods by achieving a better accuracy.

The fifth experiment is illustrating the behaviour of the accuracy when trying to obtain incremental information from new examples of the same class. We adopt an example-incremental scenario (EI), in which we train the method over streaming data providing new examples without introducing new classes. We test and compare TILDA-DA, NCM, NN, Learn++ and BRIL on CIFAR10, CIFAR100 and ImageNet50. We divide these datasets into 10 equal size parts, each part containing 5000 examples (500/50 example per class) for CIFAR10/CIFAR100, and 4500 examples (90 per class) for ImageNet50. During this experiment, incremental learning methods learn one dataset part at a time. Learn++ adds one weak classifier each time a new part is learned. Figure 4.5 shows that all methods handle example-incremental learning and improve their accuracy each time they learn new information provided by new examples. TILDA-DA consistently obtains higher accuracy than Learn++, NCM, NN and BRIL regardless of the quantity of provided data. Note that Learn++ needs large number of examples to perform, and obtains a low accuracy when only few examples are provided.

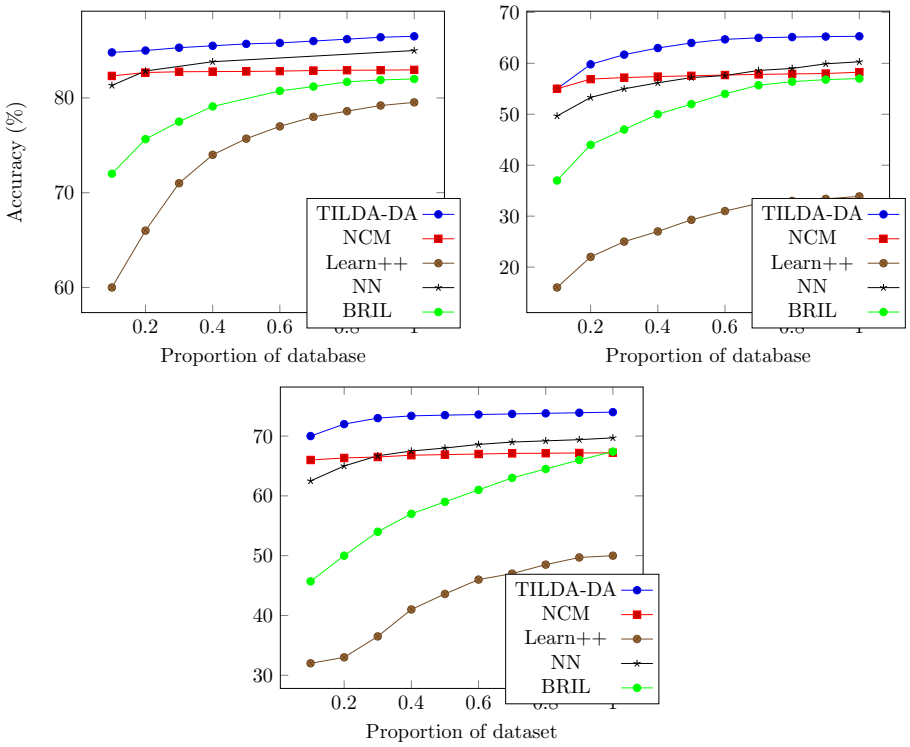


Figure 4.5: Evolution of the accuracy as a function of number of learning examples for CIFAR10 (left), CIFAR100 (right) and ImageNet50 (bottom). This figure is introduced in [7].

Table 4.2: Accuracy of TILDA on CIFAR10, CIFAR100, ImageNet50 and ImageNet ILSVRC 2012. TILDA uses the following parameters: $P = 16$ and $k = 30$. We learn incrementally one example at a time. This table is introduced in [7].

	TILDA	TILDA-DA	TILDA-NCM	TILDA-P
CIFAR100	**69.6%**	65.3%	60.7%	67%
CIFAR10	**88.7%**	86.6%	84.11%	87%
ImageNet50	**76%**	74.4%	69.2%	72%
ILSVRC 2012	**94.4%**	91%	89.6%	90%

Table 4.3: Comparison of accuracy (Acc) and memory usage (M) relative to full dataset (corresponding to 100%) for the different methods. Note that memory usage of Learn++ method represents the size of weak classifiers, and for iCaRL represents the stored feature vectors and the size of the trainable neural network. This figure is introduced in [7].

	only CI		both CI and EI					only EI
	Learn++	iCaRL	TILDA	TILDA-DA	NN	NCM	BRIL	Learn++
Acc100	34%	30%	**69.6%**	**65.3%**	60.2%	58.25%	57%	34%
M100	10.5%	8%	6%	6%	100%	**0.2%**	6%	6.8%
Acc10	79.8%	41%	**88.7%**	**86.6%**	85%	83%	82%	79.5%
M10	0.65%	2.7%	0.6%	0.6%	100%	**0.02%**	0.6%	0.65%
Acc50	54.2%	64%	**76%**	**74.4%**	69.7%	67.2%	67.4%	50%
M50	4.7%	5.6%	3.3%	3.3%	100%	**0.11%**	3.3%	3%

Table 4.3 summarizes the different incremental learning methods, and shows their obtained accuracies and memory footprints. Learn++ uses either class-incremental scenario (CI) or example-incremental scenario (EI). iCaRL performs learning process using CI. TILDA, NN, NCM, and BRIL use one-shot learning to process one example at a time providing a novel class or additional information, thus they handle both class-incremental and example-incremental at the same time. TILDA outperforms all other incremental learning methods on both accuracy and memory usage.

The last evaluation we perform aims to compare TILDA with non incremental learning methods such as TMLP and TSVM. To do so, we store and train these methods on the whole dataset. The parameters used for TILDA are $P = 16$ and $k = 30$ for CIFAR10, CIFAR100 and ImageNet50, and uses one-shot learning to process one example at a time. Table 4.4 shows that TILDA reaches an accuracy comparable to state-of-art methods, even when it learns incrementally only one example at a time.

As shown by the different evaluations, TILDA can at any instant classify data with a good accuracy (*cf.* Figure 4.4 and Figure 4.5), outperforms other incremental learning methods (*cf.* Table 4.3), and approaches non incremental state-of-art accuracy (*cf.* Table 4.4). Consequently, TILDA fulfills criterion 2.

Table 4.4: Comparison of TILDA with non-incremental learning methods. This figure is introduced in [7].

	TILDA	TILDA-DA	TMLP	TSVM
Acc (CIFAR100)	**69.6%**	65.16%	68.6%	67.6%
M (CIFAR100)	**6%**	**6%**	100%	100%
Acc (CIFAR10)	88.7%	86.6%	90%	89.2%
M (CIFAR10)	**0.6%**	**0.6%**	100%	100%
Acc (ImageNet50)	**76%**	74.4%	75.2%	75%
M (ImageNet50)	**3.3%**	**3.3%**	100%	100%

4.8 Hardware Implementation

In Section 4.6 and 4.7, we showed that TILDA fulfill all criteria introduced in Section 3.1, and thus represents a good solution to overcome Learning on Chip (LOC) problems. In this section, we exploit the simplicity of TILDA method and its good performance to propose an incremental learning on chip (ILOC) solution. We assume that a generic feature extraction is performed by an external CPU which provides feature vectors $\mathbf{X}^m$ to the FPGA. Consequently, we introduce a hardware implementation to compute only the incremental classifier part. The DNN hardware implementation can be performed using compression methods and hardware architectures introduced in Chapter 3.

4.8.1 Data Quantization

All data and signals are quantized on $n = 18$ bits fixed-point representation, which enables to use only 1 dedicated multiplier block (Xilinx DSP Block) for each multiplication. In addition, we perform local quantization by setting the number of integer bits $\tilde{n} \leq n$ at each step of the algorithm. In the subsequent figures depicting hardware blocks, we include the width of each bus in italics. The number $\tilde{n}$ of integer bits at each step of the hardware implementation changes as follows:

- Feature-vector, Anchor-vector: $\tilde{n} = 5$,
- Distance: $\tilde{n} = 10$,
- Address, Counter: $\tilde{n} = 18$,

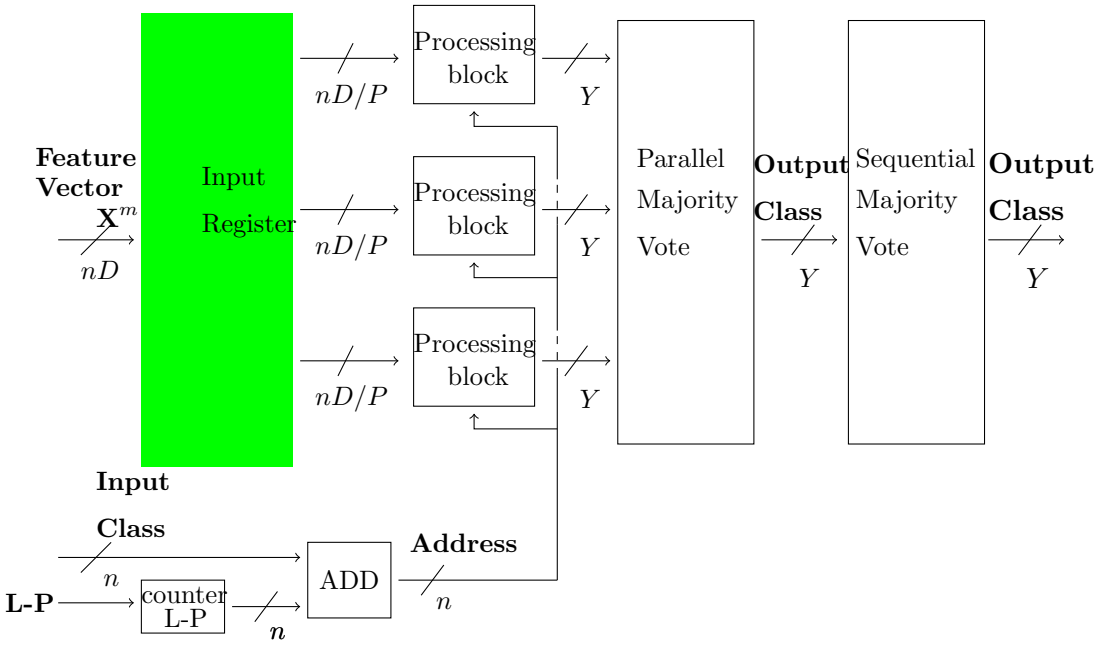


Figure 4.6: Hardware architecture for incremental learning.

- Distance×Counter: $\tilde{n} = 16$,
- Anchor-vector×Counter: $\tilde{n} = 10$,
- Anchor-vector+Feature-vector: $\tilde{n} = 10$.

4.8.2 Hardware Architecture

An overview of the hardware architecture is presented in Figure 4.6. Each input feature vector $\mathbf{X}^m$ is split into P sub-vectors, and processed on P Processing blocks in parallel. Each processing block p gets a sub-vector, as well as an address that is generated by the counter L-P block. Each processing block outputs the class associated to a sub-vector. The obtained classes $(\mathbf{y}^p)_{1\leq p\leq P}$, which represent a Y-dimensional vector, are used to compute a Parallelized Majority vote, and classify the input feature vector $\mathbf{X}^m$. Finally, Sequential Majority vote is used to output the class of the original signal when data augmentation is performed to classify unlabelled data.

Processing block

We use this component to learn or classify a sub-vector. This component has three inputs: feature sub-vector, learning-processing signal *L-P*, and address (generated by Counter/L-P) and has only one output, the obtained class of a feature sub-vector one-hot encoded on Y bits, where Y is the number of classes. Given a feature sub-vector

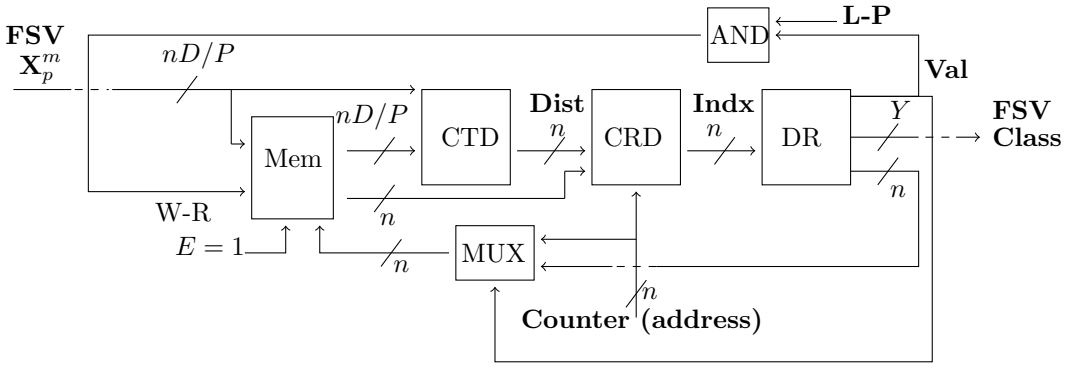


Figure 4.7: Hardware architecture of Processing block. Note that "FSV" refers to feature subvector, "Mem" refers to Memory, "CDT" refers to Compute Distance, "CRD" refers to Compare Distance and "DR" to Distance Register.

$\mathbf{X}_p^m$, we first compute the euclidean distance between $\mathbf{X}_p^m$ and $\mathbf{V}_p^i$ (where $\mathbf{V}_p^i$ is the first anchor vector addressed by the address generator), multiply the distance by anchor vector's counter, and store the result in the register r_p in Compare Distance block. We repeat the same process using each $\left(\mathbf{V}_p^j\right)_{i \leq j \leq i+k}$, compare the result with the r_p value, and store the smallest one in r_p. Finally, Compare Distance block outputs the index of the nearest $\mathbf{V}_p^j$ from $\mathbf{X}_p^m$. Given this index, Distance register block outputs the same index and the class of anchor vector corresponding to the index. It also outputs a validation signal *val*, which is equal to 1 when the nearest $\mathbf{V}_p^j$ from $\mathbf{X}_p^m$ has been determined. During the learning process (*L-P*=1), when *val* signal is equal to 1, R-W becomes 0 and we use the feature sub-vector and index from the Distance Register block through the multiplexer to modify the memory content according to Algorithm 2. The inverse values of indexes are stored in Look-up tables and multiplied by the output of the Distance Register block (*cf.* Figure 4.7).

Counter/L-P

This component is an ordinary counter, which counts from 0 to *Modulo_in* value. This counter uses a signal *L-P* which is equal to 1 during learning phase, and 0 during classification phase. This signal sets *Modulo_in* to k during learning phase, to generate only k different addresses in order to read only anchor vectors of a specific class. During classification phase, it sets Modulo_in to Yk, in order to read all anchor vectors.

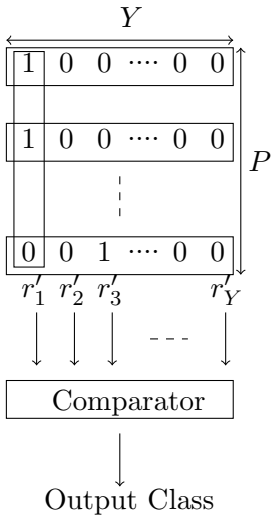


Figure 4.8: Overview of majority vote process.

Memory

The Memory block contains two memory blocks (Xilinx UltraRam technology), one to store anchor vectors (URAM A-V), and the other one to store corresponding counters (URAM Counters). Addresses are provided by Counter/L-P. It is also performs the multiplication/division of an anchor vector and its corresponding counter, and the sum between an anchor vector and an input feature sub-vector.

Majority vote

Class vectors $\mathbf{y}^p$ are one-hot encoded on Y bits. Parallel Majority Vote computes a bitwise addition over all $(\mathbf{y}^p)_{1\leq p\leq P}$ vectors. The Y results stored into $\left(r'_y\right)_{1\leq y\leq Y}$ registers are compared sequentially, and the class index y corresponding to the register r'_y with the highest value is attributed to the unlabelled feature vector $\mathbf{X}^m$ (*cf.* Figure 4.8).

Sequential Majority vote is computed only when using data augmentation. This block takes as input only one class vector $\mathbf{y}^p$ and performs an addition between each y bit of the input class vector and the y inner register. A final comparison is performed between each y results, which outputs a global predicted class vector.

During training, when Compare Distance block compares two distances, Compute Distance block computes a new distance between input feature sub-vector and another anchor vector. Thus, the learning phase needs $k + 3$ clock cycles per feature vector. Precisely, it takes k cycles to compute/compare distances, 1 cycle to multiply anchor

vector with its corresponding counter, 1 cycle to add the result with the input feature sub-vector and increment its counter and 1 cycle to divide the result by this incremented counter. During classification process, sequential majority vote needs at least S clock cycles (S represents the number of feature vectors resulting form data augmentation) to give an output, parallel majority vote needs at least YS clock cycles to classify S feature vectors, and processing block needs YkS clock cycles to classify S sub-vectors resulting from data augmentation and corresponding to the same input. In the proposed architecture, these three blocks work at the same time, thus YkS is the number of clock cycles needed to classify an unlabelled feature vector, with Yk cycles to compute distances, repeated S times to classify all feature vectors resulting from data augmentation.

4.8.3 Results

The proposed hardware architecture has been implemented and validated by software simulation over a batch of examples. We provide synthesis results of the hardware architecture on a Xilinx Ultra Scale Vu13p (xcvu13p-figd2104-1-e) Field Programmable Gate Array (FPGA) in Table 4.5. We also include synthesis results from BRIL that we proposed in [25] as a reference.

Performance estimates are given for CIFAR10 for $P = 16$, $K = 30$ and $Y = 10$, yielding an accuracy of 89.1%/87% with/without data augmentation, instead of 88.7%/86.6% obtained for 32-bit encoding. To obtain feature vectors, we use inception V3 [96] ($D = 2048$). 2048 DSPs are used to compute distances and $P = 16$ more to multiply/divide anchor vectors by their corresponding counters. Power consumption and maximum clock frequency of the whole system are estimated to about 8 Watts and 208 MHz. The estimated time needed to learn/classify an input vector is 158.2/1442 ns at maximum clock frequency, corresponding to an acceleration factor of 10^4 when compared with a software simulation delay using an I7 870 (2.93 GHz) processor.

4.9 Summary of the Chapter

In this chapter, we discussed the incremental learning concept and compared some incremental learning methods. We introduced Budget Restricted Incremental Learning (BRIL) and Transfer Incremental Learning Using Data Augmentation (TILDA), two incremental learning methods using feature extraction, vector segmentation, and majority vote. An incremental learning method is well adapted to real life tasks, and presents a good solution to perform learning on chip (LOC), since it learns only one or few ex-

Table 4.5: FPGA results for TILDA and BRIL implementations on vu13p (xcvu13p-figd2104-1-e) ($D = 2048$, $P = 16$, $K = 30$).

	TILDA	BRIL [25]
Memory usage (bits)	11059488	6553600
Look-up Tables (LUT)	152546	95654
DSP	2064	2048
Maximum frequency (MHz)	208	204
Learning delay (ns)	158.2	5
Classifying delay (ns)	1442	1470
Energy consumption (W)	7	13
Accuracy (%)	87	82

amples at a time. We also introduced a hardware architecture to perform incremental learning on chip (ILOC). Such a hardware architecture allows an embedded system to train a model on chip that dynamically adapts to new data.

A future work would be exploring further the methods for splitting feature vectors, data augmentation strategies and a weighted majority vote to improve the accuracy, and also introducing hardware architecture and implementation of the pretrained CNN to propose a complete embedded incremental learning on chip solution.

Chapter 5

Conclusion

Contents

5.1 Conclusion and Perspectives

5.1.1 Summary of the Thesis

In this Ph.D. manuscript, we tackled the problem of implementing deep learning solutions in the context of resource limited devices. We reviewed several propositions to reduce both memory and computations, using pruning, quantification, or factorization. We also introduced novel methods to handle the case of incremental learning, since a vanilla deep learning model does not have the ability to learn new information over time without destroying previously acquired knowledge.

5.1.2 Summary of Contributions

This section summaries the different contributions introduced in this thesis, and briefly explains how each one is an answer to the problematic of the Ph.D.

In Section 3.5, we introduced Shift Attention Layer (SAL), a novel attention-based pruning method that replaces a vanilla convolutional layer by the concatenation of a shift operation and a simple 1x1 convolution. The idea is to use pruning not only to reduce neural network size but also to considerably reduce the number of operations. To this end, we equipped each convolutional kernel with an attention mechanism aiming at learning which weight should be kept in the resulting shift layer. We demonstrated that SAL reduces both memory and computations required by a deep learning based inference solution, and thus may address the societal and technical challenges mentioned in the introduction since it tackles the two main limitations: memory and computations, that beget these challenges. However, SAL requires extra parameters at training stage, and as such cannot be used to accelerate the training procedure.

In Section 3.8, we presented a hardware architecture to implement SAL on FPGA. In addition to SAL, we binarized the remaining weights using BWN. As such we ended up with a simple hardware architecture that shifts a given input, and then uses a low cost multiplexer (since the weight values should be either 1 or -1). We believe that this resulting hardware implementation could be of use in many practical cases, and in particular when the time, memory or energy is limited to run a prediction. Such a solution could be in particular implemented in the context of smartphones.

In Section 3.9, we studied the effect of reducing input voltage of an embedded system implementing a deep neural network on its accuracy. We assumed that reducing input

voltage introduce errors in on chip memory where neural network parameters are stored. Therefore, we proposed to introduce the same error during training phase, causing a significant increase in the neural network accuracy under the effect of reduced input voltage. Such a contribution aims at addressing the energetic impact of deep learning.

In Section 4.4, we focused on how to improve performance of transfer learning using vector segmentation. When using transfer learning, one usually considers a pre-trained neural network on a large dataset to process a smaller dataset. Hence, one ends up with sparse feature vectors where useful information would be spread among the coordinates. Thus, splitting resulting features vectors, classifying each part independently, and finally using a majority vote to classify resulting feature subvectors can considerably increase the performance and robustness of the method. In this contribution, we aimed at addressing a scientific challenge, which is to better understand and exploit the feature vector specific distributions when relying on transfer learning with deep neural networks.

In Section 4.6, we proposed Transfer Incremental Learning using Data Augmentation (TILDA), an incremental learning method. TILDA relies on pre-trained neural networks to extract feature vector of a given input, then splits this feature vector to improve the performance, and uses a Nearest Mean Class (NCM) inspired classifier to incrementally learn one example at a time. TILDA tackles incremental learning, a real time problem and a technical challenge, where the algorithm is adapted on the fly using new data while keeping previous acquired knowledge.

In Section 4.8, we introduced a hardware architecture to implement TILDA on FPGA. Such a hardware architecture can easily fit on an FPGA due to the simplicity of TILDA algorithm, giving an incremental learning on chip solution, and aims at addressing some technical challenges. Indeed, it tackles both real time problem and learning and processing data on chip.

A main contribution of this Ph.D., that corresponded to a significant effort of research, was to list, understand, implement and compare the numerous techniques that have been introduced in the literature to tackle the problem of compressing deep learning methods. We quickly understood that this problem is missing standardized and fair benchmarks allowing to quickly grasp the main interests (and disadvantages) of proposed methods. We were very surprised to observe that many proposed techniques in the literature (some of which were cited hundreds of times at the time of writing this document) resulted in almost no gain (and sometimes even worst performance) than simply smartly tuning the hyperparameters on the initial baseline architecture. Too many papers advantageously benefit from a modest understanding of the actual specificities of GPUs or even modern processors to push methods that apparently reduce the number

of parameters or number of computations, but actually result in longer processing time and memory usage. We sincerely hope that this manuscript will help the readers better understand the effect of mainstream methods on memory and computations.

5.1.3 Perspectives

As mentioned in Section 3.10 and Section 4.9, the different contributions discussed in this manuscript can be extended in numerous ways, and used as a starting point to explore other and more efficient solutions.

Our work and other state-of-the-art methods on shift layers open a new considerable perspective. Convolutional Neural Networks were considered as the best solution that can be applied to process datasets containing images. However, in this manuscript we showed that shift layers based methods can outperform CNNs in some conditions. Indeed, a shift layer based method is able to achieve a better accuracy than CNNs while using less parameters. Moreover, it does not compute the complex convolution operation, accelerates data processing and uses less resources. Thus, such a method can be a substitution to CNNs. Shift layers have the main interest of focusing the computations to very precise kernels (made of only 1 weight each), and offer new perspectives of understanding the performance of deep neural networks.

Quantization methods introduced in this manuscript aim at reducing memory footprint and computations only for classification (or inference) phase. Since learning phase is more expensive, reconsidering these methods and using them to reduce memory and computations during training would be an important contribution in this field. This question should definitely attract more interest, as it is quite clear that many applications of deep learning will require fine tuning the parameters on the fly.

Finally, we believe that a learning on chip solution where deep learning models are trained on an embedded system with limited resources such as smartphones or FPGAs would be the next major subject. Indeed, such a solution aims at substituting GPUs by embedded systems to train neural network. It will use hardware architectures and quantization methods designed to reduce memory and computations of inference phase as a starting point and re-adapt them to propose a learning on chip solution. Thus, learning on chip will provide a cheaper solution to train neural networks on cheaper devices (smartphones or FPGAs) accessible to everyone, with a low energy consumption.

Deep learning has become a central technology of today. It is still very unclear how it is going to continue to permeate science. But it appears that compression is a key

challenge of the field. Not only compression is required for some concrete applications, but it could change how fast the field is going to advance, and how accessible it is going to be to small companies and associations. Making deep learning solutions accessible to everyone, with a lesser ecological impact, and a clearer understanding of its fundamental functioning, are contributions we would hope to participate to in the coming years.

Bibliography

[1] Arash Ardakani, Carlo Condo, and Warren J Gross. A convolutional accelerator for neural networks with binary weights. In *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pages 1–5. IEEE, 2018. 68

[2] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine learning and the bias-variance trade-off. *arXiv preprint arXiv:1812.11118*, 2018. 38

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009. 75

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006. 23, 24

[5] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):16, 2018. 76

[6] Gian Marco Bo, Daniele D Caviglia, and Maurizio Valle. An on-chip learning neural network. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 4, pages 66–71. IEEE, 2000. 76

[7] Ghouthi Boukli Hacene, Vincent Gripon, Nicolas Farrugia, Matthieu Arzel, and Michel Jezequel. Transfer incremental learning using data augmentation. *Applied Sciences*, 8(12):2512, 2018. 10, 78, 83, 89, 90, 91, 92, 93

[8] G. Chen, D. Sylvester, D. Blaauw, and T. Mudge. Yield-driven near-threshold SRAM design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(11):1590–1598, Nov 2010. 70

[9] Xiuyuan Cheng, Xu Chen, and Stéphane Mallat. Deep haar scattering networks. *Information and Inference: A Journal of the IMA*, 5(2):105–133, 2016. 30

[10] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *arXiv preprint arXiv:1003.0358*, 2010. 86

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015. 8, 40, 45, 65, 66

[12] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. 41, 45

[13] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. 30

[14] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018. 43, 45

[15] R.G. Dreslinski, M. Wieckowski, D Blaauw, D Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proc. of the IEEE*, 98(2):253–266, Feb. 2010. 69, 70

[16] Zeki Erdem, Robi Polikar, Fikret Gurgen, and Nejat Yumusak. Ensemble of svms for incremental learning. In *International Workshop on Multiple Classifier Systems*, pages 246–256. Springer, 2005. 10, 77

[17] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999. 10, 19, 76

[18] Jort F Gemmeke, Daniel PW Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 776–780. IEEE, 2017. 25, 81

[19] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014. 76

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. 23

[21] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013. 83

[22] Vincent Gripon, Ghouthi B Hacene, Matthias Löwe, and Franck Vermet. Improving accuracy of nonparametric transfer learning via vector segmentation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2966–2970. IEEE, 2018. 79, 81

[23] Ghouthi Boukli Hacene, Vincent Gripon, Matthieu Arzel, Nicolas Farrugia, and Yoshua Bengio. Quantized guided pruning for efficient hardware implementations of convolutional neural networks. *arXiv preprint arXiv:1812.11337*, 2018. 8, 40, 51, 66, 76

[24] Ghouthi Boukli Hacene, Vincent Gripon, Nicolas Farrugia, Matthieu Arzel, and Michel Jezequel. Budget restricted incremental learning with pre-trained convolutional neural networks and binary associative memories. In *Signal Processing Systems (SiPS), 2017 IEEE International Workshop on*, pages 1–6. IEEE, 2017. 78, 82, 83

[25] Ghouthi Boukli Hacene, Vincent Gripon, Nicolas Farrugia, Matthieu Arzel, and Michel Jezequel. Incremental learning on chip. In *Signal and Information Processing (GlobalSIP), 2017 IEEE Global Conference on*, pages 789–792. IEEE, 2017. 97, 98

[26] Ghouthi Boukli Hacene, Carlos Lassance, Vincent Gripon, Matthieu Courbariaux, and Yoshua Bengio. Attention based pruning for shift networks. *arXiv preprint arXiv:1905.12300*, 2019. 8, 40, 53

[27] Ghouthi Boukli Hacene, François Leduc-Primeau, Amal Ben Soussia, Vincent Gripon, and François Gagnon. Training modern deep neural networks for memory-fault robustness. 8, 10, 40, 70

[28] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep

neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 8, 40, 62, 63, 64, 65

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015. 48

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 30, 31, 33, 52, 71

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016. 70, 71

[32] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018. 8, 40, 46, 47

[33] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012. 30

[34] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 64

[35] Seunghoon Hong, Tackgeun You, Suha Kwak, and Bohyung Han. Online tracking by learning discriminative saliency map with convolutional neural network. *CoRR*, abs/1502.06796, 2015. 79

[36] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 30

[37] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 8, 40, 48, 49

[38] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 7, 2017. 71

[39] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017. 30, 33, 35, 52

[40] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 8, 40, 48, 49, 62, 65, 81

[41] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 29

[42] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011. 82

[43] Yunho Jeon and Junmo Kim. Active convolution: Learning the shape of convolution for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4201–4209, 2017. 53

[44] Yunho Jeon and Junmo Kim. Constructing fast network through deconstruction of convolution. *arXiv preprint arXiv:1806.07370*, 2018. 8, 40, 47, 52, 56, 57, 58

[45] X. Jiao, M. Luo, J. Lin, and R. K. Gupta. An assessment of vulnerability of hardware neural networks to dynamic voltage and temperature variations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 945–950, Nov 2017. 70

[46] Nikola Kasabov. *Evolving connectionist systems: Methods and applications in bioinformatics, brain study and intelligent machines*. Springer Science & Business Media, 2013. 10, 19, 76

[47] S. Kim, P. Howe, T. Moreau, A. Alaghi, L. Ceze, and V. S. Sathe. Energy-efficient neural network acceleration in the presence of bit-level memory errors. *IEEE Trans. on Circuits and Systems I: Regular Papers*, pages 1–14, 2018. 69, 70

[48] Animesh Koratana, Daniel Kang, Peter Bailis, and Matei Zaharia. Lit: Learned intermediate representation training for model compression. In *International Conference on Machine Learning*, pages 3509–3518, 2019. 64

[49] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint:

An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pages 1742–1752, 2017. 42, 43

[50] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 24

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 30, 31, 32, 33, 56

[52] Ilja Kuzborskij, Francesco Orabona, and Barbara Caputo. From n to n+ 1: Multiclass transfer incremental learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3358–3365, 2013. 78

[53] Griffin Lacey, Graham W Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*, 2016. 76

[54] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. 30

[55] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. 30, 31

[56] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 8, 40, 44, 46, 57

[57] Yangqing Li, Saurabh Prasad, Wei Chen, Changchuan Yin, and Zhu Han. An approximate message passing approach for compressive hyperspectral imaging using a simultaneous low-rank and joint-sparsity prior. In *Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS), 2016 8th Workshop on*, pages 1–5. IEEE, 2016. 8, 40, 41, 45

[58] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072–1086, 2018. 76

[59] C. Liu, M. Hu, J. P. Strachan, and H. Li. Rescuing memristor-based neuromorphic design with high defects. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. 70

[60] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 44

[61] Vincenzo Lomonaco and Davide Maltoni. Comparing incremental learning strategies for convolutional neural networks. In *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*, pages 175–184. Springer, 2016. 77

[62] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017. 8, 40, 44, 47

[63] Vladimir Macko, Charles Weill, Hanna Mazzawi, and Javier Gonzalvo. Improving neural architecture search image classifiers via ensemble learning. *arXiv preprint arXiv:1903.06236*, 2019. 25, 33

[64] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017. 24

[65] Thomas Mensink, Jakob Verbeek, Florent Perronnin, and Gabriela Csurka. Metric learning for large scale image classification: Generalizing to new classes at near-zero cost. *Computer Vision–ECCV 2012*, pages 488–501, 2012. 78

[66] Thomas Mensink, Jakob Verbeek, Florent Perronnin, and Gabriela Csurka. Distance-based image classification: Generalizing to new classes at near-zero cost. *IEEE transactions on pattern analysis and machine intelligence*, 35(11):2624–2637, 2013. 10, 77, 78

[67] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. 8, 40, 42, 44, 45

[68] José Fernando García Molina, Lei Zheng, Metin Sertdemir, Dietmar J Dinter, Stefan Schönberg, and Matthias Rädle. Incremental learning with svm for multimodal classification of prostatic adenocarcinoma. *PloS one*, 9(4):e93600, 2014. 10, 77

[69] Michael D Muhlbaier, Apostolos Topalis, and Robi Polikar. Learn++. nc: Combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes. *IEEE transactions on neural networks*, 20(1):152–168, 2009. 77

[70] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014. 79

[71] Francisco Ortega-Zamorano, José M Jerez, Daniel Urda, Rafael M Luque-Baena, Leonardo Franco, et al. Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers. *IEEE Trans. Neural Netw. Learning Syst.*, 27(9):1840–1850, 2016. 76

[72] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010. 76, 79

[73] Hyunsun Park, Jun Haeng Lee, Youngmin Oh, Sangwon Ha, and Seungwon Lee. Training deep neural network in limited precision. *arXiv preprint arXiv:1810.05486*, 2018. 42

[74] Kolin Paul and Sanjay Rajopadhye. Back-propagation algorithm achieving 5 gops on the virtex-e. In *FPGA Implementations of Neural Networks*, pages 137–165. Springer, 2006. 76

[75] Anastasia Pentina, Viktoriia Sharmanska, and Christoph H Lampert. Curriculum learning of multiple tasks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5492–5500, 2015. 77

[76] Tomaso Poggio and Gert Cauwenberghs. Incremental and decremental support vector machine learning. *Advances in neural information processing systems*, 13:409, 2001. 10, 77

[77] Robi Polikar, Lalita Udpa, Satish S Udpa, and Vasant Honavar. Learn++: an incremental learning algorithm for multilayer perceptron networks. In *Acoustics, Speech, and Signal Processing. ICASSP'00. Proceedings.IEEE International Conference on*, volume 6, pages 3414–3417. IEEE, 2000. 9

[78] Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 31(4):497–508, 2001. 9, 10, 77

[79] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. 41, 45, 65

[80] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators.

In *Proc. 43rd Int. Symp. on Computer Architecture (ISCA'16)*, pages 267–278, Piscataway, NJ, USA, 2016. IEEE Press. 70

[81] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. iCaRL: incremental classifier and representation learning. 2017. 10, 76, 78

[82] Marko Ristin, Matthieu Guillaumin, Juergen Gall, and Luc Van Gool. Incremental learning of ncm forests for large-scale image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3654–3661, 2014. 78

[83] Giorgio Roffo, Simone Melzi, and Marco Cristani. Infinite feature selection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4202–4210, 2015. 46

[84] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985. 30, 36

[85] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 25, 33, 87

[86] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 8, 40, 49, 50, 52, 65, 70, 71

[87] Jeffrey C Schlimmer and Douglas Fisher. A case study of incremental concept induction. In *AAAI*, pages 496–501, 1986. 76

[88] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016. 13

[89] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 30, 31, 32, 33, 51

[90] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 38, 72

[91] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks, 2019. 64

[92] Yu Sun, Ke Tang, Leandro L Minku, Shuo Wang, and Xin Yao. Online ensemble learning of data streams with gradually evolved classes. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1532–1545, 2016. 10, 77

[93] Nadeem Ahmed Syed, Syed Huan, Liu Kah, and Kay Sung. Incremental learning with support vector machines. 1999. 10, 77

[94] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 30, 51

[95] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. 13, 51

[96] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015. 81, 88, 97

[97] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013. 76

[98] Yichuan Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*, 2013. 30

[99] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in neural information processing systems*, pages 640–646, 1996. 76

[100] Jean-Charles Vialatte and François Leduc-Primeau. A study of deep learning robustness against computation failures. In *Proc. 9th Int. Conf. on Advanced Cognitive Technologies and Applications*, Feb. 2017. 70

[101] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7686–7695, 2018. 8, 40, 43, 45

[102] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G. Wei. A 28nm soc with a 1.2ghz 568nj/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 242–243, Feb 2017. 70

[103] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. Technical report, Stanford Univ Ca Stanford Electronics Labs, 1960. 36

[104] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. *arXiv preprint arXiv:1711.08141*, 2017. 8, 40, 47, 51, 57, 62

[105] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. *arXiv preprint arXiv:1806.09228*, 2018. 8, 40, 63, 64

[106] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang. Fault-tolerant training enabled by on-line fault detection for RRAM-based neural computing systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018. 70

[107] Kohei Yamamoto and Kurato Maeno. Pcas: Pruning channels with attention statistics. *arXiv preprint arXiv:1806.05382*, 2018. 8, 40, 46, 47, 57

[108] L. Yang and B. Murmann. SRAM voltage scaling for energy-efficient convolutional neural networks. In *18th Int. Symp. on Quality Electronic Design (ISQED)*, pages 7–12, March 2017. 70

[109] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 23–32. ACM, 2019. 76

[110] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are

features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014. 79

[111] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018. 46, 48, 57

[112] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 2019. 24

[113] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 52

[114] Hongyuan Zha, Xiaofeng He, Chris Ding, Ming Gu, and Horst D Simon. Spectral relaxation for k-means clustering. In *Advances in neural information processing systems*, pages 1057–1064, 2002. 63

[115] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018. 24

[116] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018. 49

[117] Jun Zheng, Furao Shen, Hongjun Fan, and Jinxi Zhao. An online incremental learning support vector machine for large-scale data. *Neural Computing and Applications*, 22(5):1023–1035, 2013. 77

[118] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016. 8, 40

[119] Zhi-Hua Zhou and Zhao-Qian Chen. Hybrid decision tree. *Knowledge-based systems*, 15(8):515–528, 2002. 76

[120] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016. 42, 45

[121] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 30

[122] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018. 25, 33, 34, 35

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : **Traitement et apprentissage des réseaux de neurones profonds sur puce**

Mots clés : Apprentissage profond, Compression des réseaux de neurones, Vision par ordinateur, Systèmes embarqués.

Résumé : Dans le domaine de l'apprentissage machine, les réseaux de neurones profonds sont devenus la référence incontournable pour un très grand nombre de problèmes. Ces systèmes sont constitués par un assemblage de couches, lesquelles réalisent des traitements élémentaires, paramétrés par un grand nombre de variables. À l'aide de données disponibles pendant une phase d'apprentissage, ces variables sont ajustées de façon à ce que le réseau de neurones réponde à la tâche donnée. Il est ensuite possible de traiter de nouvelles données.

Si ces méthodes atteignent les performances à l'état de l'art dans bien des cas, ils reposent pour cela sur un très grand nombre de paramètres, et donc des complexités en mémoire et en calculs importantes. De fait, ils sont souvent peu adaptés à l'implémentation matérielle sur des systèmes contraints en ressources. Par ailleurs, l'apprentissage requiert de repasser sur les données d'entraînement plusieurs fois, et s'adapte donc difficilement à des scénarios où de nouvelles informations apparaissent au fil de l'eau.

Dans cette thèse, nous nous intéressons dans un premier temps aux méthodes permettant de réduire l'impact en calculs et en mémoire des réseaux de neurones profonds. Nous proposons dans un second temps des techniques permettant d'effectuer l'apprentissage au fil de l'eau, dans un contexte embarqué.

Title : **Processing and Learning Deep Neural Networks on Chip**

Keywords : Deep Learning, Compression of Neural Networks, Computer Vision, Embedded Systems.

Abstract : In the field of machine learning, deep neural networks have become the inescapable reference for a very large number of problems. These systems are made of an assembly of layers, performing elementary operations, and using a large number of tunable variables. Using data available during a learning phase, these variables are adjusted such that the neural network addresses the given task. It is then possible to process new data.

To achieve state-of-the-art performance, in many cases these methods rely on a very large number of parameters, and thus large memory and computational costs. Therefore, they are often not very adapted to a hardware implementation on constrained resources systems. Moreover, the learning process requires to reuse the training data several times, making it difficult to adapt to scenarios where new information appears on the fly.

In this thesis, we are first interested in methods allowing to reduce the impact of computations and memory required by deep neural networks. Secondly, we propose techniques for learning on the fly, in an embedded context.