POLITECNICO DI TORINO

Department of Electronics and Telecommunications
Master's Degree in Electronic Engineering

Master's Thesis

Optimizing Deep Neural Networks on GPUs

Supervisors:
Prof. Maurizio Martina
Prof. Muhammad Shafique
Univ. Ass. Muhammad Abdullah Hanif

Candidate:
Alberto Marchisio

April 4, 2018

Acknowledgments

I would like to thank all the people who helped me in developing this thesis, with suggestions, reviews and observations. They have my respectful gratitude, even if I have the responsibility for every mistake contained in this work.
First, I would like to thank my supervisors: Professor Maurizio Martina, Professor Muhammad Shafique and University Assistant Muhammad Abdullah Hanif. They all made a huge impact, motivating and guiding me in this work. A special thank you to Prof. Martina, who provided me a solid background, thanks to the projects for the course of Integrated System Architectures and gave me the opportunity to get in contact with the Computer Architecture and Robust, Energy-Efficient Technologies (CARE-Tech) team, leaded by Prof. Shafique, who had exceptionally visionary ideas and guided me successfully step by step through all the challenges. Thank to all the people working at the Embedded Computing System Group of Vienna University of Technology, who helped me giving the necessary equipment and support. A special mention to my tutor, Univ. Ass. Hanif, with whom I have worked side by side during the six months at TU Wien and who has always been present when I needed his suggestions.
I am pleased to thank my parents too, who always sustained me, believed in me and allowed me to get where I am. And also the rest of the family, who contributed in encouraging and influencing me: grandparents, uncles, cousins. A great thank to all my friends, expecially to Stefano and Matteo, who are the people whith whom I shared my best moments, but also to the other ones, older and newer, that have enjoyed my company wherever I have been in these years of tough study.

Summary

Convolutional Neural Networks (CNN) have become the state-of-the-art in image classification since the success of LeNet-5 architecture on MNIST dataset. Recent developments have allowed to train very Deep Neural Networks (DNN) and achieve high level accuracy in computer vision task. The key of their success is the significant improvement on GPU performance and parallel computation, that allows DNNs to be trained in a decent amount of time.
On the contrary, Deep Neural Networks are very memory and computationally intensive. They are unfeasible to deploy in real time or mobile applications, where power and memory are constrained. In real applications, DNN inference usually exceeds the available resources on mobile platforms and does not meet real-time constraints, because of GPU limited bandwidth.
For this reason, finding a good tradeoff between accuracy and memory footprint is essential to achieve high efficiency in DNNs. We propose two different approaches of optimization: improve the accuracy (without significant impact on power and computations) on one side; reduce the memory occupied by the DNN model, limiting the accuracy loss, on the other side. Both approaches require an increased training time before obtaining the final model, but they achieve better results (respectively in terms of accuracy or compression) then the baseline network. Having defined the motivations of our research, we propose two methodologies for the two approaches, supported by experimental results and analyses focused on some interesting aspects.

HyCNN: Employing Multi-Type SELU and RELU Activation Functions for Hybrid CNN Architectures with Improved Accuracy
Among the various possibilities that lead to an accuracy improvement, we choose to look at the activation function of the DNN. Rectified Linear Units (RELU) are widely used in state-of-the-art DNNs, because of their simplicity. Moreover, *RELU activations,* in order to keep the parameters at the same order of magnitude across each layer, *require normalization, which is a very compute-intensive process*, but it is necessary to make the training work properly. Recently, another activation function, Scaled Exponential Linear Unit (SELU) has become very attractive, since, for a specific choice of parameters, it has self-normalizing properties. Indeed, it leads

to zero mean and unit variance, which are similar to Batch Normalization (BN) technique. The reason of this interesting property can be explained by looking at the differences between RELU and SELU: while RELU is flat for negative inputs, SELU has an exponential behavior (with nonzero derivative). This property leads to a better training process overall with respect to RELU, because all the weights are updated during backpropagation phase. Another important characteristic of SELU is its self-normalizing property, which leads to an higher accuracy. Dropout method becomes more critical when using SELU activation function, because it injects a Gaussian noise on the weights at each iteration, in such a way that it contributes to increase the variance. We can compensate this effect by setting the dropped weights to the low variance value ($\lim_{x \to -\infty} SELU(x) = -\lambda\alpha = \alpha'$). This method is called "alpha dropout".
Replacing RELU with SELU does not work optimally across every layer of the network. For this reason, in our methodology, we allow another degree of freedom: choosing different activation function for different layers. In this way, we obtain an Hybrid CNN (HyCNN) that presents multiple types of activation functions. Our methodology introduces a penalty of training time, because of increased exploration time, but it is typically assumed affordable to achieve a better DNN design. Experimental results show that for each network, our methodology generates the HyCNN version, and results are compared with respect to the original version. Keeping RELU in fully connected layers, while replacing RELU with SELU in the convolutional layers of the CNN and properly tuning the "alpha dropout" rate, we are able to achieve a relative error reduction in the range between 8% (AlexNet on CIFAR-100) and 17% for our DNN on CIFAR-10.

PruNet: Class-Blind Pruning Method for Deep Neural Networks
There are several different compression techniques and pruning criteria proposed by recent researches. We target a magnitude-based pruning method, called Class-Blind, that, despite its simplicity, lead to an interesting compression level. Class-Blind method does not put constraints based on the sparsity of each layer, but it allows non-uniform sparsity across each layer. In this way, we can achieve an optimal compression ratio, better than the other state-of-the-art magnitude-based pruning methods. We propose a general methodology, called "PruNet", that can be applied to sparsify a neural network: iteratively pruning and retraining the network, until the accuracy drops under a certain desired level. Moreover, pruning has a regularizing property, because during the first stages of pruning, the sparse networks outperforms the original one in terms of accuracy. This effect is beneficial in order to get high compression ratios, without losing accuracy. After a certain level of sparsity, however, the accuracy loss grows very rapidly: this is the best point to set the accuracy threshold.

Results from the experiments show that networks can be compressed from 63X (AlexNet on CIFAR-100) to 191X (LeNet-5 on MNIST). Further analyses of weight distributions demonstrate that Class-Blind method outperforms other magnitude-based pruned methods, like Class-Uniform one, which penalizes more heavily the layers with few weights, like for example the first convolutional layer of LeNet-5.

Table of contents

Chapter 1

Introduction

In recent years, Deep Neural Networks (DNN) have achieved a great success in many machine learning applications, like computer vision [31], speech recognition [15], natural language processing [7] and machine translation [2]. In particular, Convolutional Neural Networks (CNN) emerged as the state-of-the-art in computer vision tasks [23, 31, 42, 44]. Accuracy have improved very quickly, so as the computational complexity. The key of their success, however, is the significant improvement on GPU performance and parallel computation, that allows DNNs to be trained in a decent amount of time. Even though they can beat humans in image classification task [49], they require a huge amount of storage and memory accesses. In real applications, DNN inference is not easy to deploy because it usually exceeds the available resources on mobile platforms and does not meet real-time constraints, because of GPU limited bandwidth.

Accuracy and resource utilization are inversely proportional and a good tradeoff between them is essential to achieve a good efficiency. We propose two different approaches of optimization: improve the accuracy (without significant impact on power and computations) on one side; reduce the memory occupied by the DNN model, limiting the accuracy loss, on the other side. Both approaches require an increased training time before obtaining the final model, but they achieve better results (respectively in terms of accuracy or compression) then the baseline network.

1.1 Thesis outline

The rest of the thesis is organized in the following chapters.

Chapter 2 provides the theoretical background of Neural Networks and the algorithms that have been used in the following chapters.

Chapter 3 describes the methodology, called "HyCNN", that allows to obtain an Hybrid CNN architecture, with different activation functions. Applying this method

leads to an accuracy improvement with respect to the original CNN.
Chapter 4 explains an iterative, magnitude-based pruning method, that reduces the number of parameters of Deep Neural Networks, without losing accuracy, thanks to the retraining process.
Chapter 5 concludes the thesis and discusses future challenges.

Chapter 2

Neural Networks overview

In this chapter, we introduce the basic concepts of Neural Networks, how they work and the recent evolution of Deep Learning. Then we explore which are the challenges that have motivated our research.

2.1 Neural Network Basics

2.1.1 What is Deep Learning?

Deep Learning is a subfield of Artificial Intellignece, concerned with algorithms inspired by the brain, called Neural Networks. Figure 2.1 contextualizes Deep learning and shows the nested connection with Artificial Intelligence.

- Artificial intelligence (AI) includes every task that is performed by a computer or a machine, withut using human mind. It includes learning, reasoning and self-correction.
- Machine Learning is a type of AI that *"gives computers the ability to learn without being explicitly programmed"* - Arthur Samuel, 1959.
- Deep Learning is a subfield of Machine Learning algorithms, where Neural Networks are called "Deep" because they have a deep level of astraction and each successive layer uses the outputs of the previous layer as input.

2.1.2 Neurons

The fundamental basic block of a Neural Network is the neuron (artificial neuron). It has been inspired by the biological neuron, which has mutual connections with other neurons in the brain, but it has a general structure, as shown in Figure 2.2a:

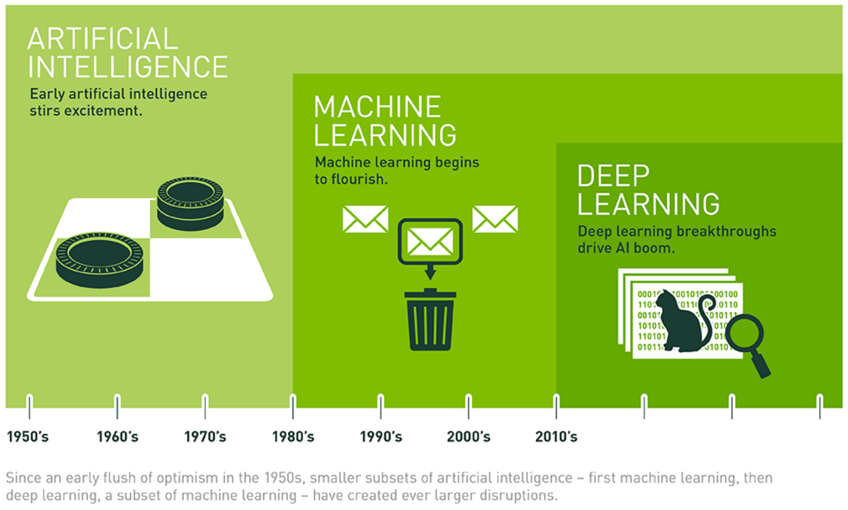
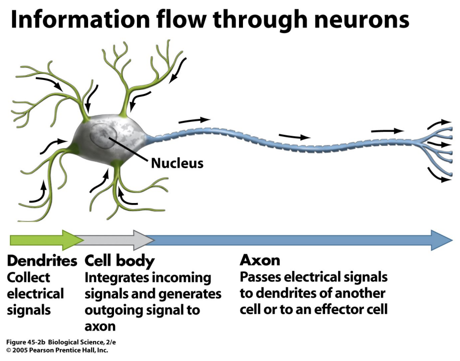
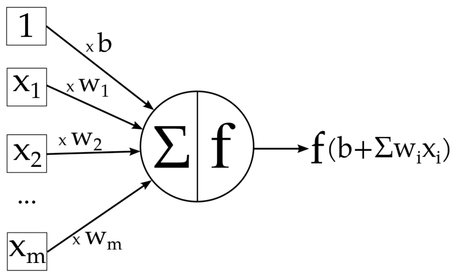

ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's
Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Figure 2.1: From Artificial Intelligence to Deep Learing. [Image source: Nvidia.]

many inputs, called dendrites, a nucleus inside the cell body and an output, called axon. The artificial counterpart (Figure 2.2b) presents many similarities, because it has several inputs, multiplied by weights, and a single output. The sum of weighted inputs and bias is propagated toward the output through an activation function.


Information flow through neurons
Nucleus
Dendrites
Collect electrical signals
Cell body
Integrates incoming signals and generates outgoing signal to axon
Axon
Passes electrical signals to dendrites of another cell or to an effector cell
Figure 45-2b Biological Science, 2/e
© 2005 Pearson Prentice Hall, Inc.

(a) Biological Neuron in a human brain.



(b) Artificial Neuron in a Neural Network. [Image source: inspirehep.net]

Figure 2.2: Comparison between a Biological and an Artificial Neuron.

The activation function, which is the key element that characterizes the neuron, can either be linear or nonlinear, depending on the application and on the position inside the network. Neurons are connected together through connections. Each connection contains a weight, that is multiplied by the signal of the connection. Connecting together many neurons, we form a Neural Network. It is composed by different layers, where the first one is called input layer, the last one output layers and in between there can be one or more hidden layers.

2.1.3 Learning

Learning (also called Training) is the process that adjusts the value of all the weights in a Neural Network. The algorithm is quite complex and in this section we are going through it.
First of all, we have to specify that there are different types of learning: supervised, unsupervised and semi-supervised. Supervised learning is the simplest type, but also the most effective with Neural Networks. The input data is labeled (it contains the information of what is the expected output) and the goal is to make the Neural Network predict correct outputs, given new inputs. In unsupervised learning, however, labels are not present: the machine learning algorithm must then find some common structures/features of inputs and automatically group them. Semi-supervised learning is a tradeoff between the two: some data is labeled, but other inputs does not contain any information.
The standard algorithm for supervised learning provides the iteration of the following steps:

1. Feed the Neural Network with the input data.
2. Predict the outputs in the forward pass.
3. Compute the cost function, depending on the predicted outputs and the desired ones.
4. Backpropagation: compute the gradient of weights with respect to the cost function.
5. Update the weights, according to the gradient computed in the backpropagation pass (gradient descent algorithm).

The dataset is typically divided into training set and test set. The training data is used in the training process, while the test data is used only in the feed-forward pass (process called inference) and do not contribute to change weight values. Since usually there are many hyper-parameters to optimize, we reserve a part of training data for validation (i. e., for hyper-parameter optimization), as shown in Figure 2.3.
A typical problem that we encounter with Deep Neural Networks is overfitting: when the training set error is much lower than the validation set error. In such scenario, the neural network has learned also some redundant features of input data and cannot generalize well on new data (validation and test set). Several

regularization methods have been proposed to reduce overfitting, including applying some pre-processing (augmentation), weight decay and dropout. In particular, dropout technique, proposed by Srivastava et al. [43], consists of randomly dropping some connections between neurons.

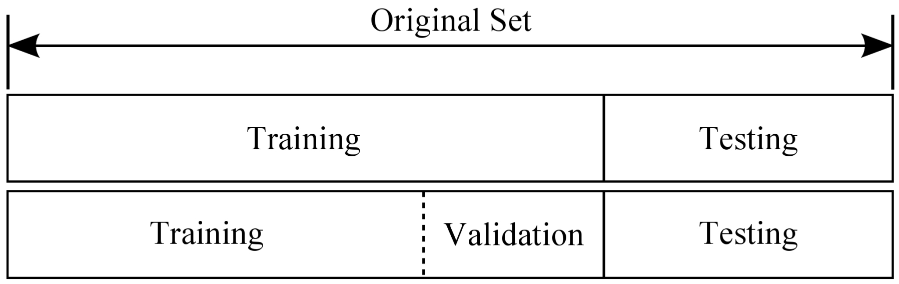


Figure 2.3: How to divide the dataset in training, validation and testing set.

Training Neural Networks is a process that requires and consumes a lot of hardware resources. Efficient hardware can improve the performances and can allow to use deeper networks in order to achieve a better accuracy. At inference time, an efficient hardware plays a key role, because it can reduce latency and energy consumptions, allowing to deploy real-time and resource-constrained applications.

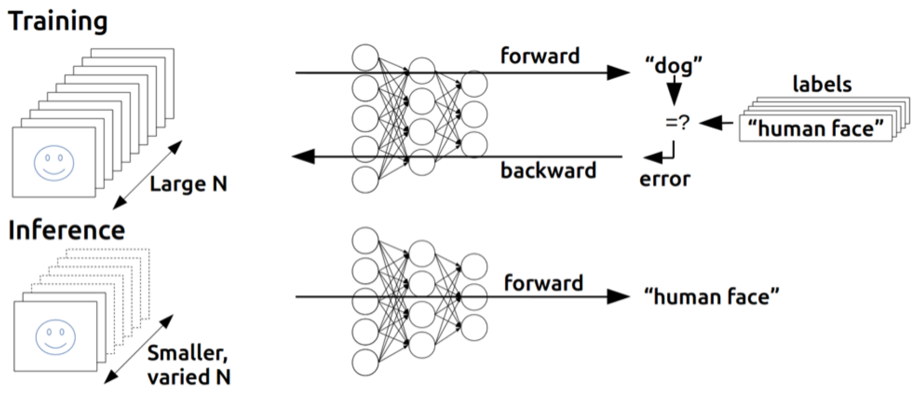


Figure 2.4: Explanation of the difference between training and inference (forward-backward pass).

Since training is much more resource-consuming than inference, a very effective compromise to get a reasonable accuracy is transfer learning: we start from a pre-trained model and we fine-tune (i. e., retrain only the last layers of the network, without updating the weights in the first layers) with another dataset. This process is made possible by the structure of the Neural Networks, where the first layers extract general features of data that can be useful also for other datasets.

2.2 Deep Neural Networks

Neural Networks that have many layers are called ”Deep”. There is not a clear distinction between a shallow and deep network, however, usually a Neural Network that counts at least eight layers is considered Deep by the community. This section gives an overview of the most used DNNs, with a particular focus on Convolutional Neural Networks (CNNs).
There are several different versions of DNNs, that can be applied to different applications: MLP (Multi Level Perceptron), CNN (Convolutional Neural Network) and RNN (Recurrent Neural Network) are among the most popular ones. A MLP is composed by fully-connected layers, where each neuron of a layer is connected to every neuron of the next layer. A CNN is particularly effective for image classification task [31], because it uses the spatial locality of input data (images) to extract features of images at different level of abstraction. A RNN captures also the temporal correlation of inputs and fits well with temporally correlated data, like in language and speech recognition.

2.2.1 CNN

Convolutional Neural Networks emerged as the state-of-the-art in computer vision tasks, since the success of LeNet-5 architecture [32], which introduced some basic concepts and features that are still useful in recent architectures, like AlexNet [31], VGGNet [42], GoogleNet [44] and ResNet [23]. A standard CNN architecture is organized with convolutional layers, followed by pooling layers and local response normalization layers. The last stages are composed by one or more fully-connected layers.

- **Convolutional layer**
  It consists on a set of filters with fixed size. Each filter is able to extract features from an image, like edges of shapes. Applying convolution means that each filter is convolved with the input: weight sharing and local connectivity are obtained.

- **Pooling layer**
  Reduce the size of the representation (subsampling), in order to reduce also the number of weights, computation speed and power consumption. The most commonly used pooling layers are average-pooling and max-pooling.

- **Local response normalization (LRN) layer**
  RELU neurons have unbounded activations and LRN is used to normalize their response. The effect of LRN is a sort of lateral inhibition. It is particularly effective in deep networks.

- **Fully-connected layer**
  Every output of the i^{th} layer is connected to every neuron of the $(i+1)^{th}$ layer.

2.3 Deep Learning Frameworks

Low-level programming Deep Neural Network for CPU or GPU execution requires a great effort. However, the operations computed by Neural Networks typically can be seen as few basic operations, like memory loads, matrix multiplications, convolutions and memory stores. These operations are already optimized for using in the mainstream efficient hardware, such as GPUs. Deep learning frameworks allow the user to write few lines of code at high level to describe training and inference algorithm for DNNs. We have used Caffe [29] and PyTorch [51] frameworks for our experiments. They both use cuDNN library, with CUDA back-end, to access to the optimized Single Instruction Multiple Data (SIMD) instructions for our GPU installed on the system, Nvidia GTX 1070.

2.4 Challenges and motivations for our research

Neural Networks are getting deeper and deeper, to achieve high accuracy. Even though they can beat humans in image classification task [49], they require a huge amount of storage and memory accesses. For these reasons, we look into inference optimizations for applications where resources are critical. In particular, we focus our research in two aspects:

1. Increase the accuracy, without increasing computations and memory requirements $\rightarrow$ Chapter 3.

2. Reduce memory footprint, without reducing accuracy $\rightarrow$ Chapter 4.

Chapter 3

HyCNN: Employing Multi-Type SELU and RELU Activation Functions for Hybrid CNN Architectures with Improved Accuracy

3.1 Introduction

Convolutional Neural Networks (CNN) are very popular among Artificial Intelligence applications, like computer vision [31], speech recognition [15] and natural language processing [7]. In particular, they emerged as the state-of-the-art in computer vision tasks [23, 31, 42, 44]. The key of their success is the significant improvement on GPU performance and parallel computation, that allows Deep Neural Networks (DNN) to be trained in a decent amount of time.
DNNs usually require nonlinear activation functions. Rectified Linear Units (RELU) are widely used in state-of-the-art DNNs, because of their simplicity. Moreover, *RELU activations,* in order to keep the parameters at the same order of magnitude across each layer, *require normalization, which is a very compute-intensive process*, but it is necessary to make the training work properly. Ioffe and Szegedy [27] proposed Batch Normalization (BN) technique, that allows to achieve a better accuracy and speedup with respect to original CNNs. Recently, Klambauer et al. [30] introduced Scaled Exponential Linear Unit (SELU) and demonstrated that, for a specific choice of parameters, it has self-normalizing properties, because it leads to zero mean and unit variance. Therefore, SELU can replace RELU activation functions in CNNs, resulting in training speedup. Moreover, SELU function has a wider learning ability

with respect to RELU, because also the negative inputs are continuously updated. As a consequence, SELU activations lead to a more efficient network in terms of accuracy.
Now a key scientific question is: *which RELU layers should be replaced by SELU in order to improve performance and accuracy?* To address the above question, we propose a novel methodology that systematically analyzes common substructures of the CNN and converges this analysis to select the best positions where to replace RELU with SELU. Through this methodology, we can obtain a Hybrid CNN that have RELU in some activation layers and SELU in others. Such HyCNN can improve the accuracy with respect to the original network. Instead of replacing activation functions in all the layers of the network, our methodology adds a degree of freedom in CNN parameters, because different types of activation functions can be selected in different layers. The final outcome is a network that has the best configuration, among all the possibilities, for each activation layer. Quantitatively, we achieve from 8% to 17% test error rate reduction in our HyCNN architectures with respect to original versions.
Section 3.2 presents the properties of state-of-the-art DNNs and SELU activation function. Section 3.3 shows a comparison between RELU and SELU and analyzes the reasons why SELU is more convenient. Section 3.4 describes our methodology. Architecture configurations and experimental results are presented in Section 3.5. Section 3.6 summarizes and concludes the work.

3.2 Related work

3.2.1 DNNs and Activation Functions

Convolutional Neural Networks have become the state-of-the-art in image classification since the success of LeNet-5 architecture [32] on MNIST dataset. Classification of much larger datasets (like ImageNet) requires an increase in the number of layers and layer size. Many years later, Krizhevsky et al. [31] introduced some new concepts regarding CNNs that have inspired further researches. Rectified Linear Units (RELU) have become a de-facto standard for activation functions, because of their advantages with respect to *tanh* or sigmoid function, as it has been shown by Glorot et al. [14]. Even if RELU does not require input normalization, a Local Response Normalization (LRN) layer is desired to aid generalization. In addition, dropout technique, widely used in state-of-the-art DNNs, inhibits overfitting.
Deeper Networks, like ResNet [23], apply very frequently Batch Normalization (BN) layers instead of LRN. As described by Ioffe and Szegedy [27], BN contributes to training speedup, because it allows to increase the learning rate and helps regularization. Therefore, backpropagation is not affected by the scale of the parameters, since

weights tends to be more stable. While LRN is applied to the output of multiple kernels belonging to the same layer, BN normalize each layer's input independently, obtaining zero mean and unit variance. As explained in Section 3.3, SELU activation function can achieve the same properties.

3.2.2 From RELU to SELU

RELU activation function, introduced by Nair and Hinton [39], has shown its great potentials in AlexNet [31] and other networks afterwards. Since it has zero derivative for negative inputs, the backpropagation error is blocked in those conditions. This is called the "dead neuron problem", because once a neuron reaches this condition, it will not escape and can be considered dead because it cannot be updated. Many researchers proposed solutions for that problem. Maas et al. [37] suggested to use Leaky RELU, where also the negative part of the activation function has a positive (linear) slope. Setting the appropriate value of the slope can be tricky, but He et al. [22] showed a method to learn the slope automatically during backpropagation. Another important direction of research is Exponential Linear Unit (ELU). Clevert et al. [6] proposed ELU, an activation function with exponential behavior in the negative part and linear in the positive one.
Another issue regarding RELU is that it introduces mean and variance shifts. This effect is accentuated when the network becomes much deep. In order to compensate it, Ioffe and Szegedy [27] implemented a new type of normalization, called Batch Normalization. BN represents the key of success for ResNet [23] and gives the main contribution to their accuracy improvement over the previous state-of-the-art DNNs. Every activation function described above require BN layer to work properly. A recent work made by Klambauer et al. [30], however, showed that Self-Normalizing Neural Networks (SNN) have the intrinsic property to automatically converge to zero mean and unit variance, without requiring explicit Batch Normalization. They propose to use Scaled Exponential Linear Units (SELU) as activation function, instead of RELU. *In our work, we show how RELU can be systematically replaced by SELU activation functions at the selected layer of a given DNN, in order to increase the training performance, also in DNNs without BN layers in their original versions.*
Chen et al. [4] and Harmon et al. [20] proposed to change activation functions in CNNs, obtaining accuracy improvement. They introduced another degree of freedom, allowing to choose a different activation function for each neuron of the same layer. Although the accuracy improves, their approach cannot avoid completely branch divergence, when the network is trained on a Single Instruction Multiple Data (SIMD) based architecture, like GPU: since each neuron can be different from the others in the same layer, Convolutional kernels are not spatially equal, then the SIMD utilization-efficiency is not optimal.

Our work differentiates from it because it introduces the possibility to change activations in different layers selectively, obtaining a Hybrid CNN, without degrading the performance. Computational efficiency is maintained because each neuron belonging to the same layer uses the same type of activation function.

3.3 SELU vs. RELU

RELU is a nonlinear activation function, expressed by Equation (3.1). It is very simple to implement and the computational effort is minimal.

$$RELU(x) = \begin{cases} 0, & if \;\; x \leq 0 \\ x, & if \;\; x > 0 \end{cases} \tag{3.1}$$

SELU, however, is a more complex function, described in Equation (3.2). This function introduces two new parameters, λ and α. They can be seen as two new hyper-parameters of the network, because their values can affect significantly the training process. In our experiments we follow the work made by Klambauer et al. [30] and use their values, reported in Equations (3.4) and (3.5), in order to achieve zero mean and unit variance.

$$SELU(x) = \lambda \begin{cases} \alpha \exp^{x} - \alpha, & if \;\; x \leq 0 \\ x, & if \;\; x > 0 \end{cases} \tag{3.2}$$

Figure 3.1(a) shows the differences between the two functions. While RELU is flat for negative inputs, SELU has an exponential behavior. This property leads to a better training process overall with respect to RELU, because all the weights are updated during backpropagation phase. In other words, the dead neuron problem is avoided. The main advantage of SELU with respect to RELU can be seen in the derivative (Figure 3.1(b)). While, for positive inputs, the derivatives are pretty similar (except for the vertical shift introduced by the parameter λ), SELU introduces a nonzero derivative also for negative inputs. That property allows to back-propagate the cost function toward the entire network. On the contrary, RELU activations block backpropagation in their negative part.
Another important characteristic of SELU is its self-normalizing property, that leads to the main motivation of our work: demonstrate that using SELU activation functions leads to a better accuracy with respect to the original network. Klambauer et al. [30] presented significant improvements when applying their proposed SNN. We propose a more general approach to select which is the best architecture configuration. We allow to choose a different activation function (RELU or SELU) on each layer. The methodology and selection decisions are explained in more detail in section 3.4. Klambauer et al. [30] proposed initialization and dropout techniques that fit well

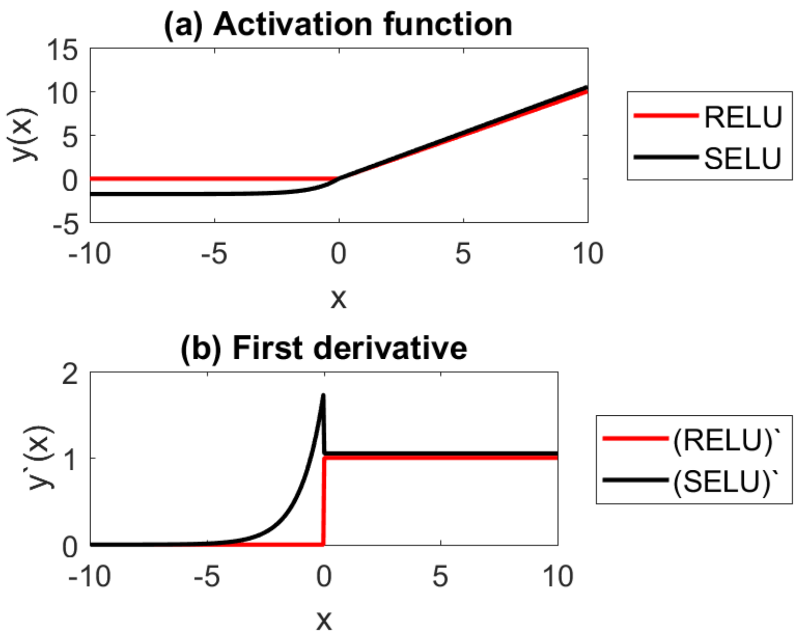


Figure 3.1: Comparison between RELU and SELU functions.

with SELU. We adopt these methods in the following sections, after having discussed further about dropout in Section 3.3.1. In addition, we analyze the consequences due to SELU of another hyper-parameter, the learning rate, in Section 3.3.2.

3.3.1 Dropout method

Dropout technique has been introduced by Srivastava et al. [43], in order to improve the regularization and to avoid CNN overfitting. They are widely used in the most common state-of-the-art networks because of their interesting properties and simple applicability with RELU activation functions. He et al. [22] proposed a weight initialization method that is efficient for RELU activations, because it limits the variance. Kingma et al. [38] analyzed how variance changes when dropout is applied. Klambauer et al. [30] revised it and proposed a new initialization method and a new dropout technique, specific for SELU. Weights are initialized in such a way that mean $E(w_i) = 0$ and variance $Var(w_i) = 1/n$, where n is the number of inputs. This methods leads to the global variance (sum of all variances of each weight in the same layer) equal to 1. For example, each weight of a 100 input layer should be initialized as a gaussian variable with zero mean and variance equal to 0.01.
Standard dropout is critical, because it injects a Gaussian noise on the weights at each iteration, in such a way that it contributes to increase the variance. Dropout

is well matched with RELU, because setting the weights to 0 (the default value) corresponds to the low variance value. Therefore, dropout works fine with RELU in order to preserve mean and variance. SELU, though, is a different function, because the low variance value corresponds to $\lim_{x \to -\infty} SELU(x) = -\lambda\alpha = \alpha'$. For this reason, Klambauer et al. [30] proposed the technique called "alpha dropout", which sets dropped weights to α', in order to preserve mean and variance. Hence, in the following sections, we adopt standard dropout when applied to RELU activations and "alpha dropout" when dealing with SELU ones. A similar approach has been adopted by Hendrycks and Gimpel [25], when dropout is applied in networks with batch normalization.

3.3.2 Learning rate analysis

The learning rate is a key hyper-parameter for CNNs and tuning its value implies large changes in accuracy and training speedup. Moreover, different types of activation functions respond in a different way, according to their first derivatives. As reported in Figure 3.1 (b), SELU derivative has a peak equal to the product $\lambda\alpha$, when $x \to 0^-$. This behavior can cause an unwanted exploding gradient problem. In other words, it is possible that for a small range of learning rate values, RELU activation can learn well, but SELU activation can suffer from exploding gradient. This effect can be compensated in two ways: either reducing the learning rate applied to SELU or introducing "alpha dropout" layers after SELU, as explained in Section 3.3.1. The second option is preferable, because the former one results in an accuracy reduction, since the learning rate is no more optimal.
Another remarkable aspect is the following: since SELU avoids dead neuron problem, training performs better than RELU when the learning rate is in middle/small range. Usually CNNs are trained with multi steps of decreasing learning rate, in order to achieve speedup with respect to a fixed rate. Using SELU activations, weights are updated even with a small learning rate, while using RELU the update is less significant. This effect is even more evident in shallow CNNs [32]. This consideration have been proved with an example using LeNet-5 on MNIST dataset, reported in Section 3.5.2.

3.4 Methodology

The proposed methodology works for any type of DNNs, from very shallow nets to the deeper ones. It introduces additional training time due to exploration, but it is considered affordable to achieve a better DNN design in terms of accuracy. Figure 3.2 shows the steps to follow in a generic architecture, with an example applied on AlexNet.

First of all, we have to identify common substructures of the network. Looking at the details of each layer, we can subdivide them into groups with similar features. A first (coarse grain) approach consists of grouping together convolutional (CONV) layers and fully-connected (FC) layers. CONV layers in DNNs, however, do not have all the same configuration. Looking for instance at AlexNet architecture (Figure 3.3), we can refine our structure if we consider also where Local Response Normalization (LRN) is applied and where not. AlexNet architecture has LRN only in the first two CONV layers. In this way, we can divide the layers in three groups:

- Group A: CONV1 & CONV2 $\rightarrow$ RELU+LRN
- Group B: CONV3, CONV4 & CONV5 $\rightarrow$ RELU
- Group C: FC6 & FC7 $\rightarrow$ RELU+DROPOUT

From now on, we process each group as a single entity, so we can analyze step by step all the possible modifications of the network, obtaining a Hybrid CNN (called HyCNN) that presents multiple types of activation functions.
In parallel to that, we have to define all the possible configurations that can be used in each group. The rectangle box of Figure 3.2 on the right lists the possibilities for AlexNet.
Starting from group A, we find out which is the best replacement by exploring one by one all the possible configurations. Based upon experimental analysis, we select the configuration that leads to a better accuracy. Some practical rules can be applied to speedup parameter selection: keep LRN layers and change dropout rate after SELU if the accuracy gets lower. A more detailed analysis about dropout method after SELU ("alpha dropout") is reported in Section 3.3.1.
Once the best configuration for the first group of layers has been found, we move on to the next group, until all the groups have been processed. In this way, we can obtain an hybrid CNN, where some layers have RELU activation function and some others have SELU. Empirically, we can assert that SELU in CONV layers leads to an higher accuracy, while FC layers perform better with RELU.
An example of the modified version of AlexNet, called HyCNN AlexNet, is shown in Figure 3.4. Compared to Figure 3.3, the modifications are minimal, but the accuracy improvement, reported in the last column of Table 3.2, is significant.

3.5 Experiments

We apply our methodology to two different CNNs, using different datasets, AlexNet on CIFAR-100 and LeNet-5 on MNIST. After having analyzed the results, we generalize our methodology as a systematic approach and validate it with two CNNs (a Custom CNN and AlexNet) on another dataset (CIFAR-10).

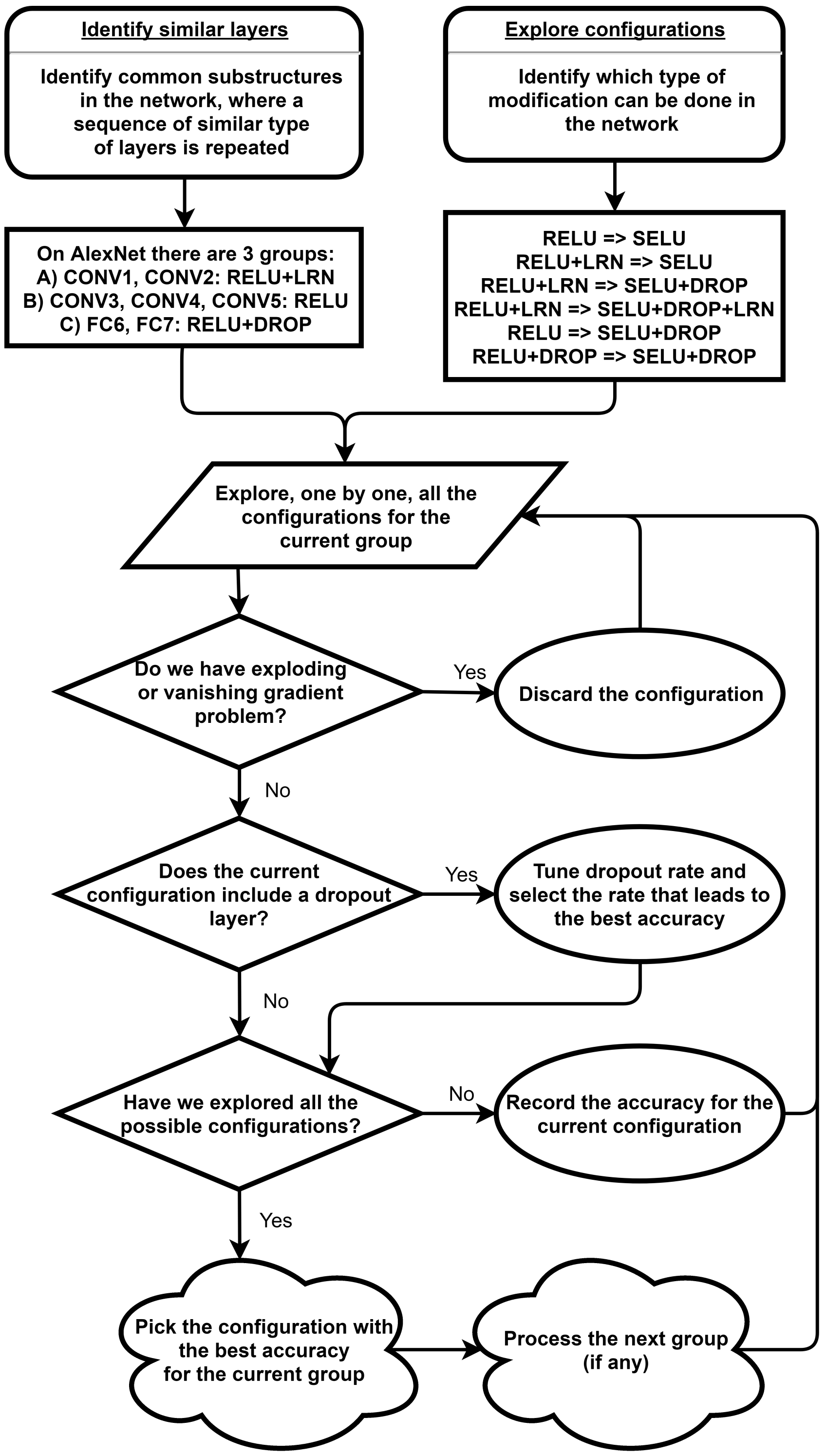

Identify similar layers
Identify common substructures in the network, where a sequence of similar type of layers is repeated
Explore configurations
Identify which type of modification can be done in the network
On AlexNet there are 3 groups:
A) CONV1, CONV2: RELU+LRN
B) CONV3, CONV4, CONV5: RELU
C) FC6, FC7: RELU+DROP
RELU => SELU
RELU+LRN => SELU
RELU+LRN => SELU+DROP
RELU+LRN => SELU+DROP+LRN
RELU => SELU+DROP
RELU+DROP => SELU+DROP
Explore, one by one, all the configurations for the current group
Do we have exploding or vanishing gradient problem?
Yes
Discard the configuration
No
Does the current configuration include a dropout layer?
Yes
Tune dropout rate and select the rate that leads to the best accuracy
No
Have we explored all the possible configurations?
No
Record the accuracy for the current configuration
Yes
Pick the configuration with the best accuracy for the current group
Process the next group (if any)

Figure 3.2: Workflow of general methodology, applied to AlexNet.

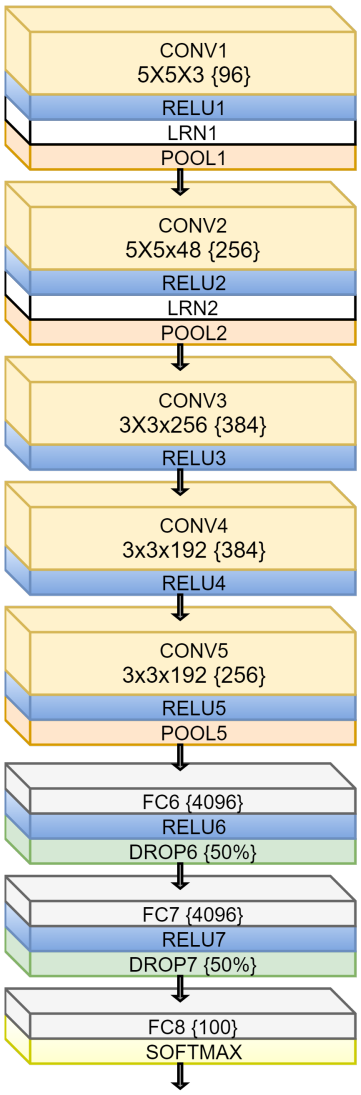


Figure 3.3: Original AlexNet architecture

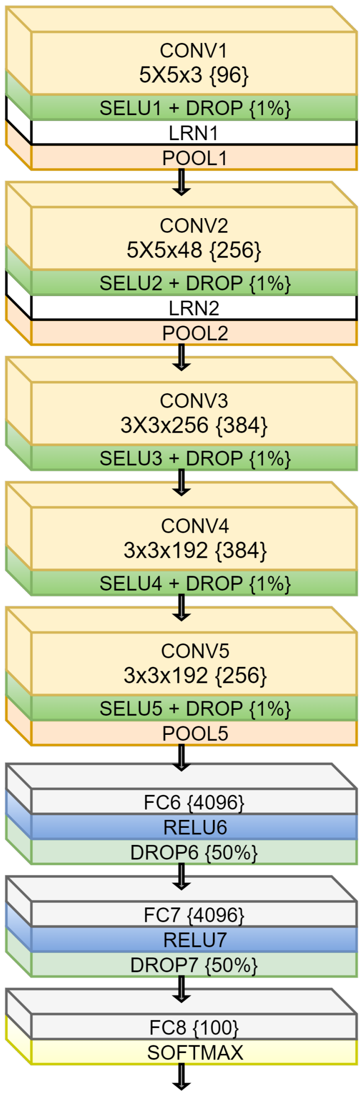


Figure 3.4: HyCNN AlexNet architecture

We use Caffe framework [29] for training and inference, performed on Nvidia GTX 1070 GPU. Its specs are reported in Table 3.1 and the experimental setup, from SW to HW is summarized in Figure 3.5. Accuracy results for different networks and datasets are reported in Table 3.4. The relative error has been computed as in Equation (3.3):

$$\epsilon_r = \frac{acc_{HyCNN} - acc_{Original}}{100\% - acc_{HyCNN}} \tag{3.3}$$

Parameters λ and α of SELU layers have been selected according to the procedure explained by Klambauer et al. [30]. Their values are reported in Equations (3.4) and (3.5).

$$\lambda = 1.050700987 \tag{3.4}$$

$$\alpha = 1.67326324 \tag{3.5}$$

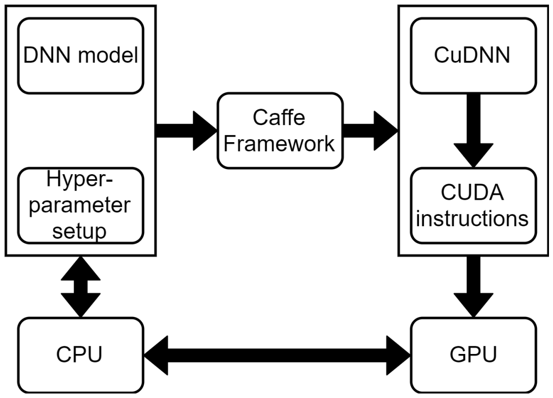


Figure 3.5: SW to HW setup

NVIDIA GTX 1070 specs	
CUDA cores	1920
Memory	8 GB DDR5
Mem. interface width	256-bit
Mem. bandwidth	256 GB/s
Single precision Flops	6.5 TeraFLOPS
Power requirement	150 W

Table 3.1: GPU specs

3.5.1 AlexNet on CIFAR-100

CIFAR-100 dataset consists of 100 classes containing 600 images each, 500 for training and 100 for testing. Evaluation have been made using ”fine” labels. Since each image has size 32x32, while AlexNet was designed for 224x224 images by Krizhevsky et al. [31], the first CONV layer has been modified in order to compensate it: instead of kernel size 11x11 and stride 4, we apply a kernel with size 5x5 and stride 1.
We trained it for 70000 iterations using a batch size of 100, with momentum = 0.9 and weight decay = 0.0005. The initial learning rate of 0.005 has been scaled by a factor 0.1 after 30000 and 60000 iterations. No preprocessing was applied to the images.
Clevert et al. [6] reported 45.80% test error rate on original AlexNet for CIFAR-100, while we achieve 45.39%.
Following our methodology, we obtained many different versions of AlexNet. The results, in terms of how TOP1 accuracy is evolved, are shown in Figure 3.6. Version configurations are described in Table 3.2. Final TOP1 accuracy values of the most significant versions (original and best one, called HyCNN) are reported also in Table 3.4. Our HyCNN AlexNet has SELU activation functions in all 5 CONV layers (followed by ”alpha dropout” with rate 1%) and RELU in FC layers. It achieves a relative error reduction of 8.12% with respect to the Original AlexNet.

Note that empirically other choices imply worst results: not applying dropout after SELU in CONV layers can lead to the exploding gradient problem, and lower accuracy is obtained if we remove LRN layers. Dropout rate tuning is essential in order to achieve an accuracy improvement.

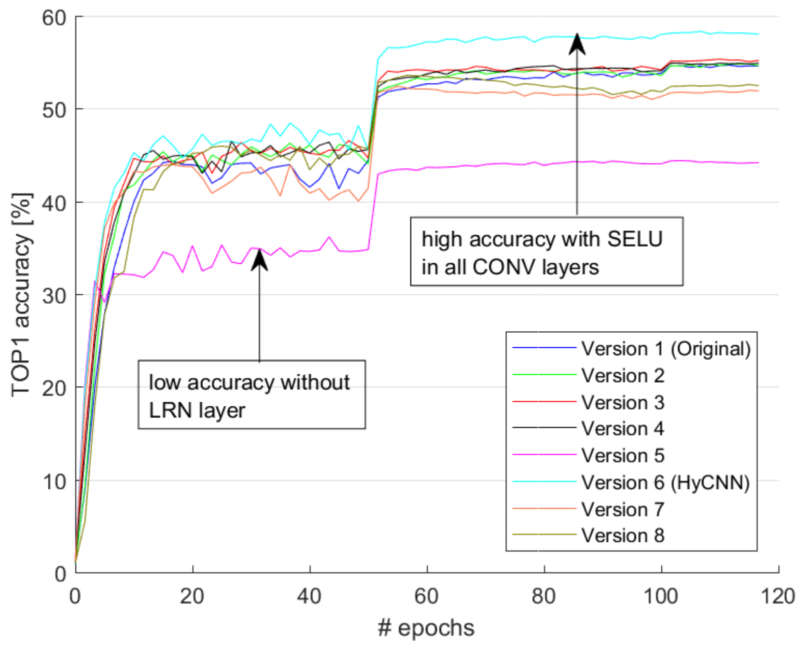


Figure 3.6: Test TOP1 accuracy of AlexNet versions on CIFAR-100 dataset.

VERSION	GROUP A	GROUP B	GROUP C	TOP1 ACC.
1 (Original)	RELU+LRN	RELU	RELU+DROP(50%)	54.61%
2	SELU+DROP(3%)+LRN	RELU	RELU+DROP(50%)	54.63%
3	SELU+DROP(1%)+LRN	RELU	RELU+DROP(50%)	55.20%
4	SELU+DROP(0.3%)+LRN	RELU	RELU+DROP(50%)	54.89%
5	SELU+DROP(1%)	RELU	RELU+DROP(50%)	44.17%
6 (**HyCNN**)	SELU+DROP(1%)+LRN	SELU+DROP(1%)	RELU+DROP(50%)	**58.02%**
7	SELU+DROP(1%)+LRN	SELU+DROP(1%)	SELU+DROP(50%)	51.91%
8	SELU+DROP(1%)+LRN	SELU+DROP(1%)	SELU+DROP(80%)	52.47%

Table 3.2: Different versions of AlexNet for CIFAR-100. CONV1 and CONV2 belong to group A; CONV3, CONV4 and CONV5 to group B; FC6 and FC7 to group C.

3.5.2 LeNet-5 on MNIST

MNIST dataset consists of a collection of handwritten digits (size 28x28), divided into 10 categories. The set consists of 60000 training images and 10000 test ones.

LeNet-5 architecture is described by LeCun et al. [32]. We trained it for 30000 iterations using a batch size of 64, with momentum = 0.9 and weight decay = 0.0005. The initial learning rate of 0.05 has been scaled by a factor 0.5 after 10000, 20000, 25000 and 29000 iterations. No preprocessing was applied to the images.
Our HyCNN architecture achieves 9.86% relative error reduction with respect to the original network. It has SELU activations (without dropout) in the CONV layers, and RELU in FC. Instead of reporting a graph similar to Figure 3.6, we report an analysis based on how the training process changes when the CNN is trained with a different number of epochs. Figure 3.7 shows how the accuracy gap between Original and HyCNN LeNet-5 varies with respect to a different training effort in terms of number of epochs. Note that those values do not refer to intermediate steps of training, but distinct training processes with a different setup of learning rates, as reported in Table 3.3.

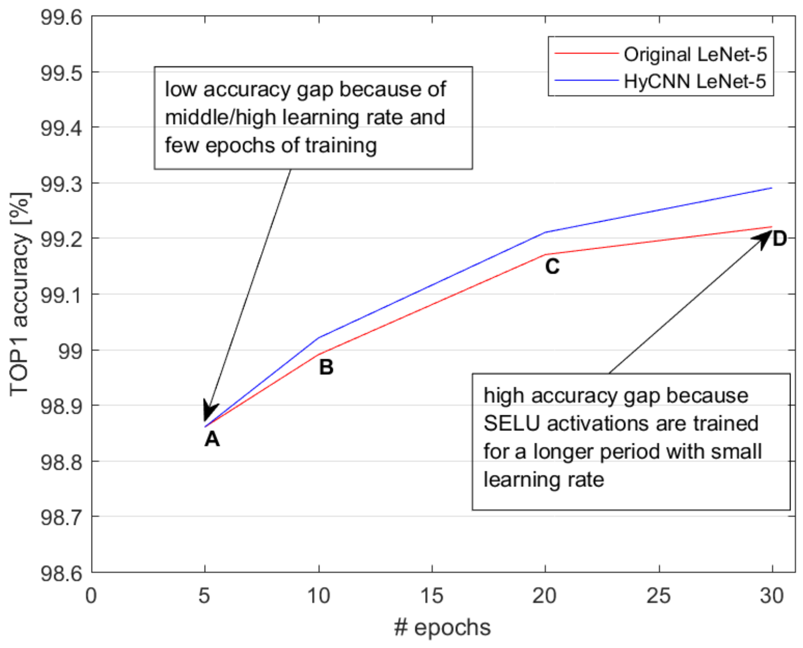


Figure 3.7: Accuracy gap between Original LeNet-5 and HyCNN LeNet-5, with respect to different number of epochs.

3.5.3 Deriving Key Observations and Design Rules

Analyzing the results obtained in Sections 3.5.1 and 3.5.2, we can obtain a systematic approach that can be applied also to other CNNs:

Setup	# epochs	Init. learn. rate	Scal. factor	Epochs steps	Final acc.
A	5	0.01	0.9	3, 4, 4.5	Original: 98.86% HyCNN: 98.86%
B	10	0.01	0.9	5, 6, 8, 9	Original: 98.99% HyCNN: 99.02%
C	20	0.05	0.5	6, 12, 16, 19	Original: 99.17% HyCNN: 99.21%
D	30	0.05	0.5	10, 20, 25, 29	Original: 99.22% HyCNN: 99.29%

Table 3.3: Setups A to D for LeNet-5 training on MNIST: for each setup, the table shows number of epochs, initial learning rate, scaling factor, number of epochs where the learning rate have been scaled and TOP1 final test accuracy.

- Replace RELU with SELU in CONV layers.
- If necessary, apply "alpha dropout" after SELU, properly tuning the dropout rate in order to achieve the highest accuracy. Note that the dropout rate is not a parameter that can be set to a certain value for every CNN, but it is strongly affected by network architecture and type of dataset.
- Maintain RELU activations in FC layers.

Following this procedure, we will obtain what we call HyCNN: an hybrid Convolutional Neural Network with multi-type activation functions (SELU in CONV and RELU in FC), that improves the accuracy over the original CNN with RELU in every layer.
Results are reported in Table 3.4. The last column of the table shows the relative error reduction obtained by our HyCNN architectures, computed using Equation (3.3).

3.5.4 Applying the Methodology to our DNN and AlexNet for CIFAR-10

CIFAR-10 dataset consists of 10 classes of labeled images with size 32x32. The set is composed by 50000 training images and 10000 test ones. Based on the analysis made using the first two experiments, we designed a custom DNN, visible in Figure 3.8, with 4 CONV layers and 3 FC layers. We trained it for 120000 iterations using a batch size of 100, with momentum = 0.9 and weight decay = 0.0005. The initial learning rate of 0.005 has been scaled by a factor 0.5 after 30000, 60000, 80000, 90000, 100000 and 110000 iterations. No preprocessing was applied to the images.
Following the design rules explained in Section 3.5.3, we replaced RELU with SELU in all 4 CONV layers and searched the optimal value of dropout rate, which is 10% in this case. Our HyCNN (Figure 3.9) reduces the relative error of 17.35%.

We repeated again the procedure with AlexNet [31], whose architecture is the same as Figure 3.3, except for FC8 that has only 10 outputs. We trained it for 70000 iterations using a batch size of 100, with momentum = 0.9 and weight decay = 0.0005. The initial learning rate of 0.005 has been scaled by a factor 0.1 after 30000 and 60000 iterations. No preprocessing was applied to the images.
Clevert et al. [6] reported 18.04% test error rate on original AlexNet for CIFAR-10, while we achieve 16.84%.
Our Hybrid architecture (HyCNN AlexNet) has SELU activations (with 10% dropout) in all 5 CONV layers and achieves 14.25% relative error reduction with respect to the original network.

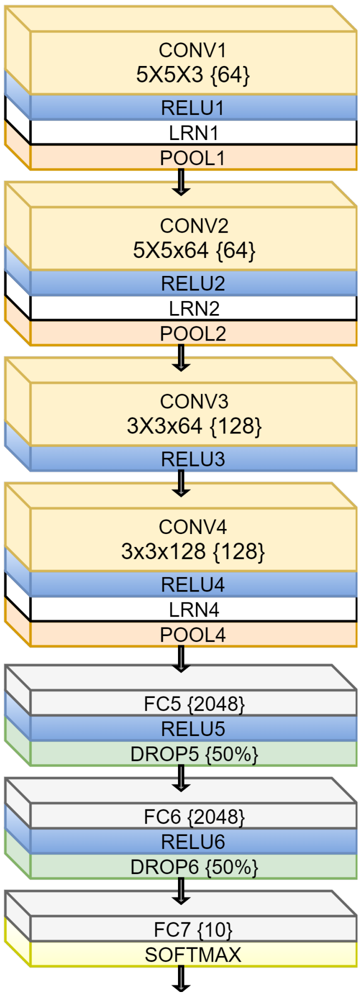


Figure 3.8: Original architecture of our DNN

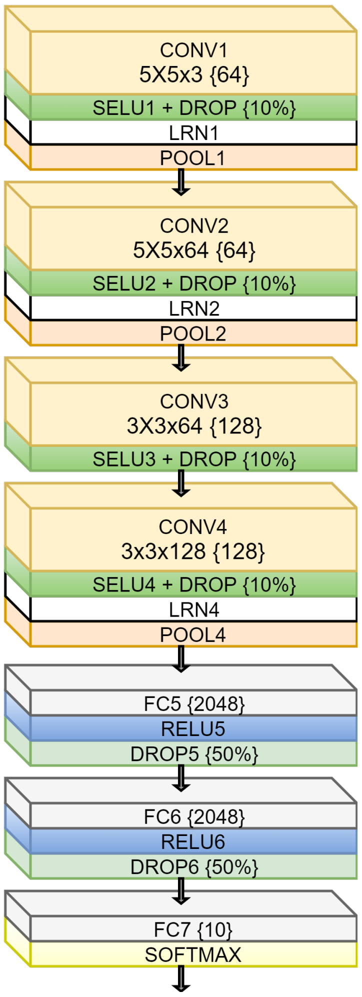


Figure 3.9: HyCNN architecture for our DNN

Dataset	Network	TOP1 Accuracy	Relative error
CIFAR-100	Original AlexNet	54.61%	-
CIFAR-100	HyCNN AlexNet	58.02%	-8.12%
MNIST	Original LeNet-5	99.22%	-
MNIST	HyCNN LeNet-5	99.29%	-9.86%
CIFAR-10	our DNN (Original)	79.30%	-
CIFAR-10	our DNN (HyCNN)	82.36%	-17.35%
CIFAR-10	Original AlexNet	83.16%	-
CIFAR-10	HyCNN AlexNet	85.26%	-14.25%

Table 3.4: Experiment results in terms of accuracy improvement and relative error reduction.

3.6 Conclusions

In this work, we have presented a systematic methodology to improve CNN accuracy using Multy-Type activation functions. In particular, our approach fits well for generic CNNs, like our proposed Custom DNN (Figures 3.8 and 3.9), that achieves more than 17% TOP1 relative error reduction on CIFAR-10. Good results have been obtained using SELU activation functions in CONV layers and RELU in FC layers. Since our methods fits well with shallow and middle range depth CNN, a possible future analysis can be performed on deeper CNN with more complex Datasets. A further exploration analysis could cover also DNNs that already have BN layers in their original versions.

Chapter 4

PruNet: Class-Blind Pruning Method for Deep Neural Networks

The work described in this chapter have been accepted for publication at the 2018 International Joint Conference on Neural Networks.

4.1 Introduction

Recent developments have allowed to train very Deep Neural Networks (DNN) and achieve high level accuracy in computer vision [23, 31, 42, 44]. Even though they can beat humans in image classification task [49], they require a huge amount of storage and memory accesses. In real applications, *DNN inference is not easy to deploy, since it usually exceeds the available resources on mobile platforms and does not meet real-time constraints*, because of GPU limited bandwidth. For this reason, many researchers in deep learning community have focused on DNN compression and efficiency improvements. There are numerous variants of compression techniques, that include different pruning methods and reduced precision representation.
We propose a magnitude-based pruning method, called Class-Blind, as described by See et al. [41]. *Our methodology requires many iterations of pruning and retraining, but we are able to achieve a significant compression with respect to the baseline model.* As a side effect, pruning has a regularizing property, because during the first stages of pruning, the sparse networks outperforms the original one in terms of accuracy. The process of iteratively pruning and retraining is effective, because it allows to progressively adjust the parameters and maintain a good level of accuracy. Our methodology has been tested both on a simple task (digit recognition on MNIST dataset, with quite simple networks, like LeNet-5 and LeNet-300-100 [32]) and on more complex configurations (VGG-16 net [42] on CIFAR-10, VGG-16 [42], AlexNet [31] and GoogleNet [44] on CIFAR-100). If we combine this work with an

efficient coding for sparse networks, like CSC or CSR, we can achieve similar benefits as reported in [18]. We do not discuss such coding methods in our work.
An overview of our work is shown in Figure 4.1. The rest of the chapter is organized as follows: in Section 4.2 we recall and summarize the work by other researchers on the same field subject; Section 4.3 explains what are the motivations of our research and the reasons why we are choosing to work on this topic; in Section 4.4 we present our methodology, that represents the fundamental aspect of our contribution; Section 4.5 describes an example, which provides us useful intuitions; the experiment section (Section 4.6) reports the most significant results; Section 4.7 concludes the article.

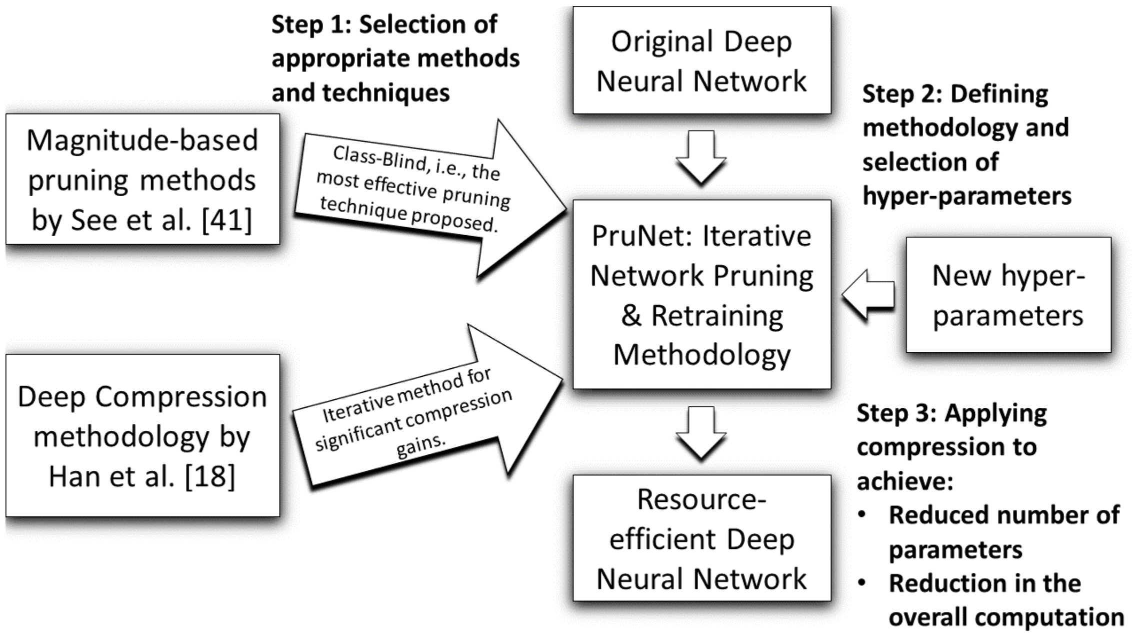

Step 1: Selection of appropriate methods and techniques
Original Deep Neural Network
Step 2: Defining methodology and selection of hyper-parameters
Magnitude-based pruning methods by See et al. [41]
Class-Blind, i.e., the most effective pruning technique proposed.
PruNet: Iterative Network Pruning & Retraining Methodology
New hyper-parameters
Deep Compression methodology by Han et al. [18]
Iterative method for significant compression gains.
Step 3: Applying compression to achieve:
• Reduced number of parameters
• Reduction in the overall computation
Resource-efficient Deep Neural Network

Figure 4.1: General overview of our work.

4.2 Related work

Making DNN inference manageable on mobile applications has become a very attractive topic and a large variety of ideas have been proposed. Vanhoucke et al. [46] propose a fixed-point implementation, achieving a considerable speedup. Other works include low-rank approximations for weight matrices (Denton et al. [11], Jaderberg et al. [28]), weight sharing (Chen et al. [5], Han et al. [18]), reduced precision (Lin et al. [35], Courbariaux et al. [10], Gupta et al. [17], Lin et al. Venkatesh et al. [47]) or even binary weights (Lin et al. [34], Courbariaux et al. [9], Courbariaux and

Bengio [8]). Hinton et al. [26] proposed the 'distillation' technique, that leads to the teacher-student approach. These techniques are orthogonal to connection pruning, as explained by Han et al. [18], where a good compression rate have been obtained. There exist several varieties of pruning methods. Structured or channel pruning (Anwar et al. [1], Li et al. [33], Wen et al. [48], He et al. [24]) allows to achieve a significant inference speedup. Recently, Changpinyo et al. [3] demonstrated also other benefits of channel-wise sparsity, but the accuracy drop limits its deployment in practice. Based on Optimal Brain Surgeon (Hassibi et al. [21]), Dong et al. proposed a layer-wise pruning approach [12] that leads to a significant compression ratio, but it requires the computation of reverse Hessian matrix. Molchanov et al. [38] introduced a new way to introduce sparsity in DNNs, called "variational dropout". The original dropout technique, introduced by Srivastava et al. [43] is modified in a way that some connections are permanently removed from the network. Inspired by this work, Louizos et al. [36] applied successfully variational dropout, obtaining an efficient coding scheme. Recently, Federici et al. [13] combined this work with Soft Weight Sharing (Ullrich et al. [45]), obtaining state-of-the-art values of compression rates.
Another widely used pruning procedure is the so called magnitude-based method: Han et al. [19] obtained a good compression ratio without sacrificing the accuracy and demonstrated in [18] that pruning can be successfully combined with other compression techniques, like weight sharing, quantization and Huffman coding. Zhu and Gupta [50] showed that, having the same memory footprint, a large sparse model (obtained by pruning and retraining) outperforms the small dense one in terms of accuracy. The main drawback of this approach is the increase of time and computation during training. In order to overcome this effect, Narang et al. [40] proposed a way to prune the network during training, with the purpose of limiting the overall computational effort during the training phase. However, this method leads to an accuracy loss and the final sparsity which can be obtained is far from the ones achieved with other approaches. Another important result has been obtained by Guo et al. [16]: they were able to significantly compress the model without loss in accuracy. Their approach is more complex than the other ones, because they exploit two operations: pruning and splicing. The splicing operation is effective because it allows to restore some important connections that were erroneously pruned in the previous step. A deeper analysis on magnitude-based methods was made by See et al. [41]. They showed that there are different pruning schemes leading to different results, while in general, despite their simplicity, magnitude-based pruning methods are effective also on Neural Machine Translation applications. A more detailed explanation of these methods is reported in Section 4.3.2.

4.3 Motivations

DNN pruning has become a hot topic nowadays because of its efficiency and it is becoming widely used in everyday applications for improving the performance efficiency of inference in embedded platforms. Our contribution is to propose a simple and efficient method to reduce the memory requirements during inference in mobile applications, where the memory occupied by the model and the power consumption are an issue, as well as real-time constraints. For this reason, the overhead introduced by an iterative approach of retraining, as explained in the work of Han et al. [19], has been considered less relevant. Iteratively pruning and retraining the network implies a much more intensive computational effort with respect to the standard training. However, this process can be performed in large data-centers, provided with powerful GPUs and optimized accelerators for Deep Learning, in such a way that time and power consumed are still acceptable.
Since the number of parameters in a network is directly proportional to the number of computations at the inference stage, an effective and commonly used parameter to evaluate pruning methods is the Compression Ratio (CR: the ratio between the number of parameters in the original model and in the sparse model, respectively), as described in Equation (4.1).

$$CR = \frac{\#\ parameters_{original\ model}}{\#\ parameters_{sparse\ model}} \tag{4.1}$$

Consequently, once we have obtained a sparse structure, we can use the same storing methods proposed by Han et al. [18], like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC). Moreover, pruning represents only the first step of DNN compression: works made by Han et al. [18] and Federici et al. [13] showed that other compression techniques are orthogonal to pruning and can be applied in further stages.
Our contribution focuses on an efficient way to apply pruning, based on Class-Blind method, that outperforms the results obtained by Han et al. [19], without a relevant accuracy loss. A detailed description of the Class-Blind method, compared to other magnitude-based methods, is reported in Section 4.3.2.

4.3.1 Efficiency of magnitude-based pruning

Among all DNN compression methods analyzed in Section 4.2, we choose to use a magnitude-based pruning method. Despite the effectiveness in terms of CR of methods based on Variational Dropout by Molchanov et al. [38] and Bayesian Compression by Louizos et al. [36], the complexity introduced by considering weights as a distribution makes these approaches difficult to deploy in everyday applications.

On the contrary, the results obtained by Han et al. [18,19] on AlexNet [31] and VGG-16 [42] trained on ImageNet dataset look very attractive for what concerns accuracy. However, some details of the pruning process (threshold value, number of retraining iterations) performed by Han et al. have not been revealed in their publications. This is another important motivation for our work: trying to obtain result comparable with (or better than) Han et al. [19], using a clearer methodology. Moreover, See et al. [41] showed as a side effect that magnitude-based pruning introduces a regularizing effect. That is the reason why low pruned models outperform their respective baseline models. As a consequence, our idea is to further prune and retrain the network (in an iterative way) beyond the level where the accuracy increases, in order to maximize the compression ratio, while keeping the accuracy loss acceptable.

4.3.2 Efficiency of Class-Blind method

See et al. [41] gave a great contribution to our work, because they analyzed three different magnitude-based pruning schemes. Despite they applied those schemes on Neural Machine Translation application only, their concepts are the key for our work, because they can be applied also to other applications, like image classification. They based the differences on the concept of "class", which means a group of neurons performing similar operations. We revisited that concept for Feed-Forward Neural Networks and we assume that each class can be seen as a weighted layer of the network. In the work by See et al. [41], the concept of class was a little bit different, because it is related to Recurrent Neural Networks for Neural Machine Translation applications. In the followings, we refer to class and layer as synonyms.
The three pruning schemes that were presented by See et al. [41] are the following:

1. Class-Distribution (CD): select threshold T, common for every layer, and compute the standard deviation σ of each layer. Then, for each layer, prune parameters below σT. This method is used by Han et al. [19]. The threshold T is equal across each layer, but the product σT can vary. *Finding the optimal value of T can be tricky.*

2. Class-Uniform (CU): select a certain percentage x and, for each layer, prune the smallest x% parameters. Easier to implement than CD. In this way pruning is equally distributed among all the layers. However, *the accuracy drop is not minimized.*

3. **Class-Blind (CB):** select a certain percentage x and prune the smallest x% parameters, regardless of which layer they belong to. In this way some layers are pruned more than others. This method is adopted by See et al. [41], showing that *it is the most efficient among these three.*

Based on the results obtained by See et al. [41], we decided to use the Class-Blind method, not only because of its simplicity, but also because it outperforms the other schemes.

4.4 Methodology

We now propose a general methodology, called "PruNet", that can be applied to sparsify a neural network. We do not describe the baseline training procedure, since it is out of the scope of our work. Then, we start from a pre-trained model of the network, i.e., already trained over the same dataset that will be applied for the pruning phase.
Figure 4.2 summarizes our methodology in a schematic way. First of all, we set the hyper-parameters. Some intuitions about how to choose these values are explained in Section 4.5.1. We reduce the initial learning rate with respect to the one used for the baseline training (e.g., if the baseline initial learning rate is 0.01, the initial one in each retraining phase could be 0.003). If the network contains some dropout layers, the dropout rate must be changed according to the rule proposed by Han et al. [19]:

$$D_r = D_o\sqrt{\frac{C_{ir}}{C_{io}}} \tag{4.2}$$

where D_r is the dropout rate during retraining, D_o the original dropout of the baseline training, C_{io} and C_{ir} are the number of connections of layer i for the original network and the sparse one, respectively. The number of training epochs can be scaled as well. While Han et al. [19] retrain convolutional layers and fully-connected layers separately, we perform retraining on the whole network at the same time.
Another group of hyper-parameters to set is composed by the ones introduced for our purpose: pruning percentage, maximum acceptable accuracy loss and pruning iterations. We have to fix only the first two values, since the number of pruning iterations depends on the other two. We choose to set the pruning percentage to a mid-range value, according to the trade-off reasons explained in Section 4.5.1. In this way, at each pruning iteration, the number of weights is reduced. The choice of the maximum accuracy loss is delicate, because it depends on the application. However, as shown in Figure 4.3, the curves become very steep toward high MSR. As a consequence, the most appropriate value lies around the focal point of an exponential fitting curve (black line of Figure 4.3).
Then, the iterative process starts. For each iteration we perform the following operations:

1. Sort the weights of the whole network in ascending absolute value order and mark the X% lowest ones according to the pruning percentage.

2. Apply a mask for each layer: a 0 corresponds to a pruned weight, while a 1 stands for a not pruned weight.
3. Retrain the (sparse) network, then compute the accuracy loss.

Finally, if the accuracy loss is still below the threshold, loop back to the next iteration, otherwise the process is terminated. Note that the final model is not the one at the last iteration, but the previous one. Since we exit from the loop when the accuracy is not acceptable, we have to restore the model at the previous step, so as to comply with the constraint.

4.5 Case study of a simple task

This section will give some useful intuitions that have been essential to formulate the more generic methodology described in Section 4.4, as well as a comparison between our approach and the Class-Uniform method. For this purpose, we train and prune LeNet-5 (developed by LeCun et al. [32]) on MNIST dataset. The network is quite shallow, because it has only two convolutional layers and two fully-connected layers, but it has been referenced as example throughout the literature very frequently. MNIST dataset is a collection of handwritten digits, grouped in 10 categories. It contains 60000 training images and 10000 test images.
We first trained the dense network, like described by LeCun et al. [32], obtaining an accuracy of 99.13 %. Hardware and software setup is reported in Section 4.6.1. Afterwards, several pruning experiments, varying some parameters, have been performed.

4.5.1 Intuitions

The pruning process is delicate, because removing connections from the network implies an accuracy loss. However, as successfully explained by Han et al. [19], the retraining process is fundamental to recover from the errors. They suggest an adjustment for the dropout rate, since the sparsity introduces itself another form of regularization. Indeed, the dropout rate should be changed according to that.
Another important hyper-parameter to consider is the learning rate. Usually, it is not constant during the whole training process, but it decreases after a certain amount of training epochs. At the first training step, all the weights and biases of the network are initialized as their default value. The pruning process, however, does not change the remaining parameter values. That is equivalent to retrain the sparse network from a pretrained model. As a consequence, the starting learning rate during retraining can be lower that the respective value set for the first train. As a demonstration of this intuition, See et al. [41] propose to divide by 2 the starting

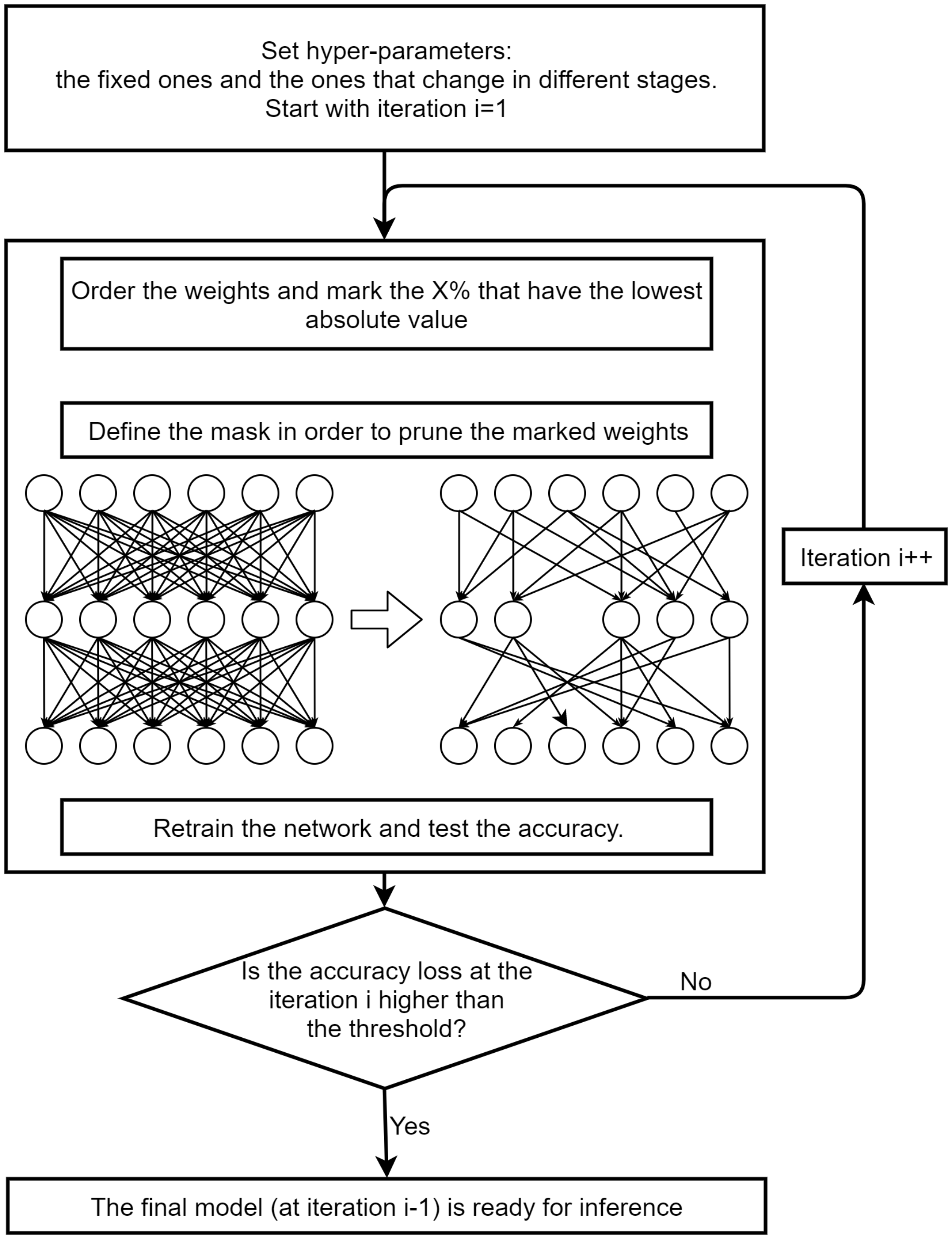

Set hyper-parameters:
the fixed ones and the ones that change in different stages.
Start with iteration i=1
Order the weights and mark the X% that have the lowest absolute value
Define the mask in order to prune the marked weights
Retrain the network and test the accuracy.
Iteration i++
Is the accuracy loss at the iteration i higher than the threshold?
No
Yes
The final model (at iteration i-1) is ready for inference

Figure 4.2: Summary scheme of methodology.

learning rate of the original model. Selecting that value is not an easy task and cannot be generalized for every setup, because it depends on the network and the dataset. Moreover, at each retraining iteration it is possible to further tune this value. Since the retraining process has some similarities to transfer learning (i.e., the network is not trained from scratch), the total number of epochs during retraining can be reduced with respect to the ones adopted for the baseline training.
Using Class-Blind pruning method, a new hyper-parameter has been introduced: the pruning percentage. It represents how many weights are going to be pruned away permanently from the network. Han et al. [19] demonstrated that iterative pruning (and retraining) is more efficient than doing the same procedure in a single iteration. In order to avoid possible terminology misunderstanding, we refer to pruning percentage as *the amount of parameters that are going to be pruned at each iteration* (e.g., after 50% pruning at the first iteration, if we prune again with a pruning percetage equal to 50%, we obtain 25% parameters with respect to the original model). It is worth noting that pruning percentage should not be confused with the Compression Ratio (Equation (4.1)). If we neglect the overhead due to a sparse memory coding scheme, the CR can be equaled to the Memory Saving Ratio (MSR): the ratio between the amount of memory occupied by the original model and the amount required by the sparse model (Equation (4.3)).

$$MSR = \frac{mem_{original\ model}}{mem_{sparse\ model}} \tag{4.3}$$

Choosing this value have a huge impact on the success of our procedure, since it defines the ability of the network to recover the accuracy from the error introduced by pruning. It has implications also on retraining time / retraining epochs, while their respective relations are quite complex. Empirically, we can assume that:

- A low value of pruning percentage allows a lower retraining effort to restore the accuracy, but requires high expense in training time due to more iterations.

- A faster approach can be obtained using an high value of pruning percentage, but it implies higher damages to the network, that cannot always guarantee an optimal recovery at the retraining stage.

According to the aforementioned considerations, a middle-range value looks more attractive. At each iteration, the value could either remain constant or change. While it is simpler to keep the pruning percentage constant at every iteration, a possible design could be starting with high value in the first stages and progressively decreasing it.
Since our final goal is to maximize the compression ratio, while maintaining an

acceptable accuracy, we define another parameter, the Accuracy Loss (AL, Equation (4.4)), which is the relative difference of accuracy between the sparse and the original model.

$$AL = \frac{Acc_{sparse\ model} - Acc_{original\ model}}{Acc_{original\ model}} \tag{4.4}$$

Considering acceptable, for example, an Accuracy Loss of 0.1%, we are able to set the number of pruning (and retraining) iterations such that the accuracy loss stays below that threshold. Thanks to the regularizing effect of pruning, explained by See et al. [41], for relatively low values of MSR, the Accuracy Loss reaches negative values, while at a certain point it grows exponentially. The behavior is shown explicitly in Figure 4.3 for our current setup (LeNet-5 on MNIST). The graph looks very promising, because it outperforms by around a factor 10 the results obtained by Han et al. [18] on a similar approach. This has been obtained by running simulations, using Class-Blind method, with different (constant) pruning percentages, from 40% to 90%, and setting the threshold of AL equal to 0.1%. Overall, the figure shows that for low MSR, the sparse network outperforms the baseline, due to the regularizing effect introduced by pruning. The accuracy loss remains quite stable, until a certain point (around a MSR of 100X) where it starts growing exponentially. In the first part the pruning percentage is not strictly relevant, since the accuracy loss lies in the range [-0.2,0]. For higher MSR, however, the difference is more significant: while for high pruning percentages the Accuracy Loss grows very quickly, a lower pruning percentage allows to better recover from that error, thus pushing the MSR to the maximum.

Figure 4.3, however, shows only the general behavior of the network, while many aspects cannot be seen directly. Introducing a magnitude-based sparsity implies removing weights that have a low absolute value. For this reason, showing the weight distribution can be useful to understand the pruning process and what actually happens inside the neural network. Figures 4.5a to 4.5l show the weight distribution of our reference network, LeNet-5 (architecture reported in Figure 4.4), at each pruning stage. Every figure shows a separate distribution for each layer of the network, where L1 and L2 stand for the first two convolutional layers, and L3 and L4 stand for the third and fourth (fully-connected) layers. Each graph has been obtained by subdividing the weights in intervals, counting the weight values belonging to each interval and plotting histograms. Each red curve represents the fitting normal distribution.

In the first figures, until the seventh pruning and retraining stage (Figure 4.5h), the total number of parameters is progressively decreasing, but at the same time, for every layer, there is a clear peak centered at 0. Thus the large majority of weights have small absolute value. Starting from the eighth stage (Figure 4.5i), the

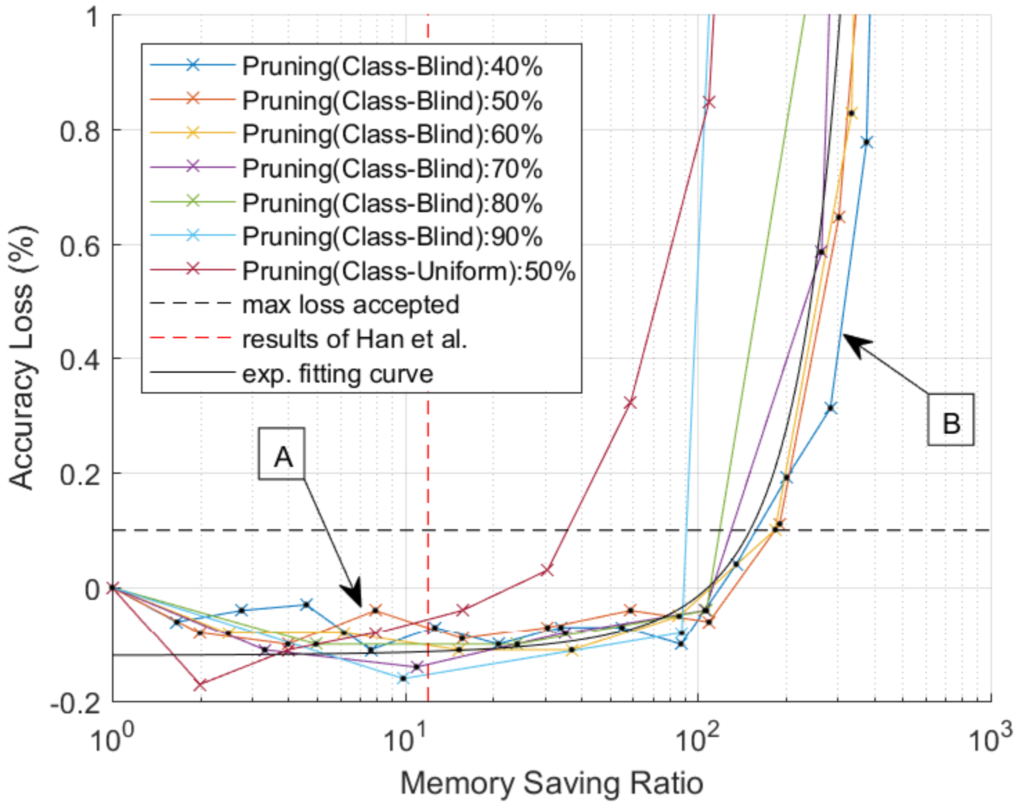


Figure 4.3: Accuracy Loss with respect to Memory Saving Ratio in LeNet-5. Box A: accuracy is slightly improved because pruning+retraining has a regularizing effect. Box B: accuracy drops significantly because the number of parameters becomes too low.

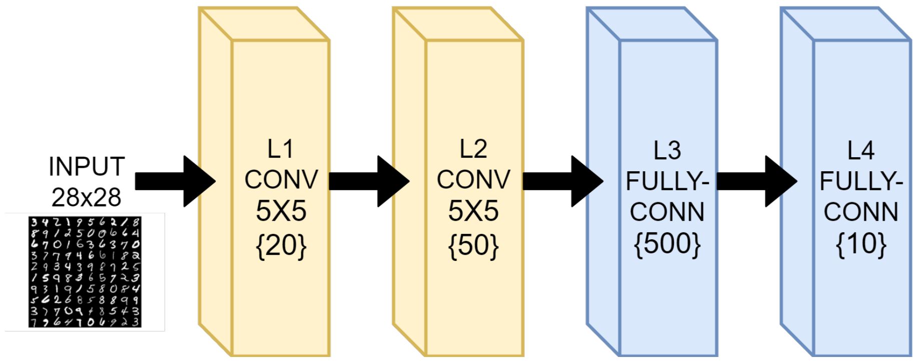
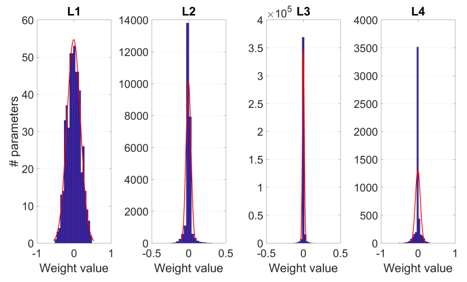
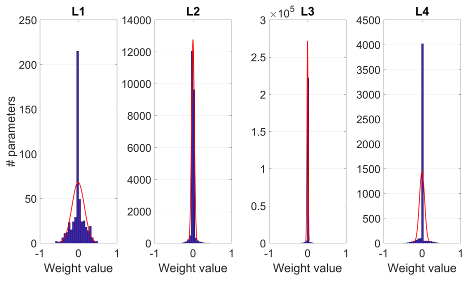


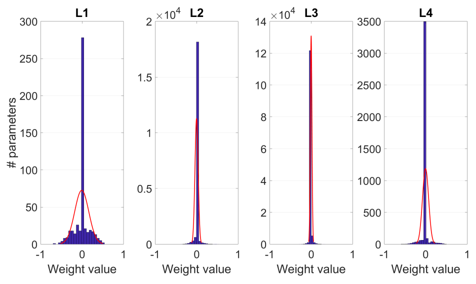
Figure 4.4: LeNet-5 architecture for our experiments.



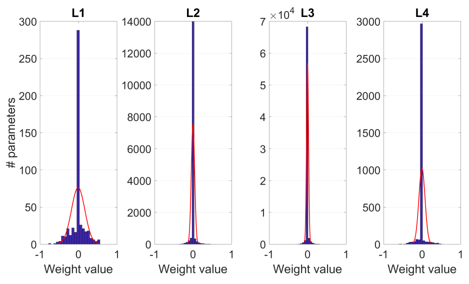
(a) After original training phase.



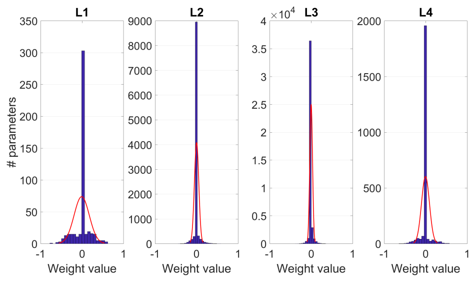
(b) After the 1st stage, using CB method.



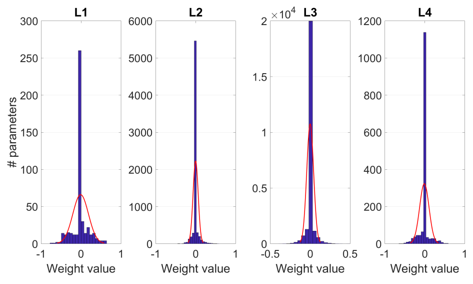
(c) After the 2nd stage, using CB method.



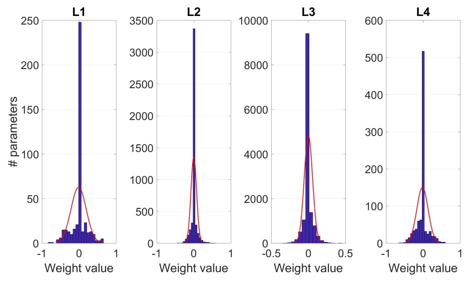
(d) After the 3rd stage, using CB method.



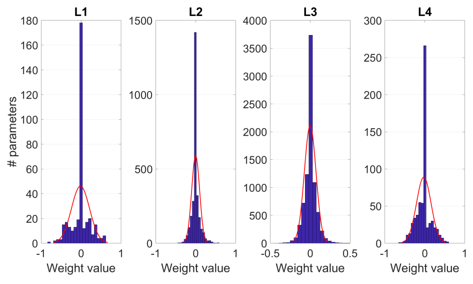
(e) After the 4th stage, using CB method.



(f) After the 5th stage, using CB method.

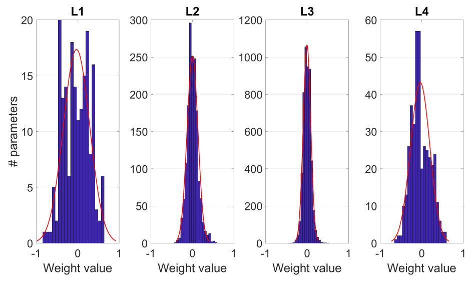


(g) After the 6th stage, using CB method.

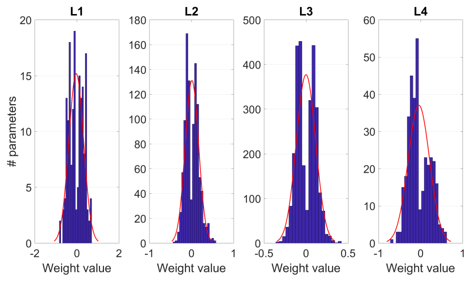


(h) After the 7th stage, using CB method.

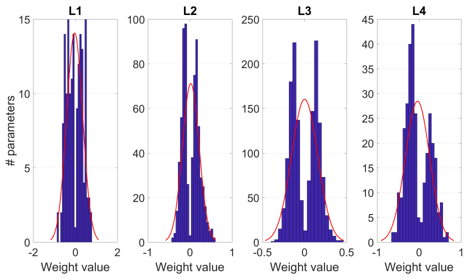
Figure 4.5: Weight distribution of LeNet-5, across different pruning & retraining stages, using the Class-Blind method.



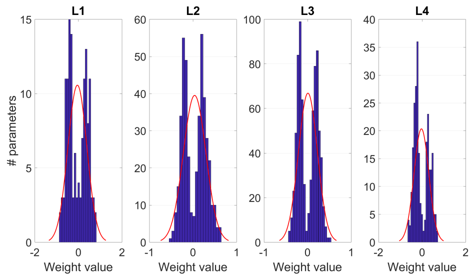
(i) After the 8th stage, using CB method.



(j) After the 9th stage, using CB method.



(k) After the 10th stage, using CB method.



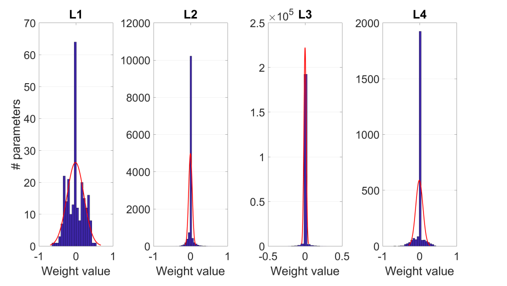
(l) After the 11th stage, using CB method.

Figure 4.5: (continued)Weight distribution of LeNet-5, across different pruning & retraining stages, using the Class-Blind method.

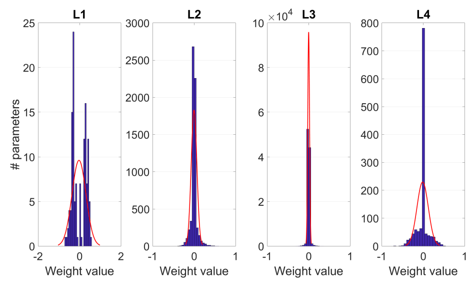
distributions are becoming smoother. Finally, in the last stages, the pruning effect introduces a hole around zero. This result means that the network is approaching its limit, because the retraining procedure is not able to recover completely from the error generated by pruning. In other words, in the last stages, few weights are remaining. Then, also the weights with middle range value are pruned. As a drawback, the accuracy is getting lower.

4.5.2 Class-Uniform vs. Class-Blind

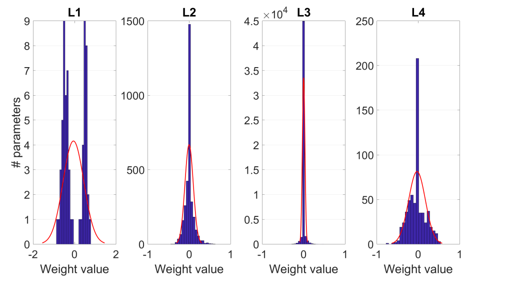
Similar experiments, setting the pruning percentage to 50%, have been carried on using Class-Uniform method. This approach is less flexible, because it does not allow to have sparsity unbalance across layers. Figure 4.3 shows also a comparison between the two methods at the same conditions (pruning percentage of 50%): it is clear that Class-Blind method outperforms the Class-Uniform one, since in the latter configuration, the Accuracy Loss start growing rapidly after the 5th iteration (MSR = 31), while the former one allows to reach the 8th one (MSR = 191). The reason of that behavior can be explained again looking at the weight distribution graphs. Looking at Figure 4.6, in particular at Figure 4.6e, that represents the weight distribution across each layer of LeNet-5 after the fifth iteration, it is evident that sparsity penalizes very heavily the layer 1, that contains few parameters. On the contrary, layer 3, that contains the majority of weights, could have been sparsified more, as with Class-Blind method in Figure 4.5f.



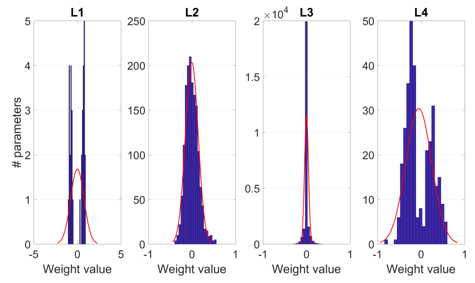
(a) After the 1st stage, using CU method.



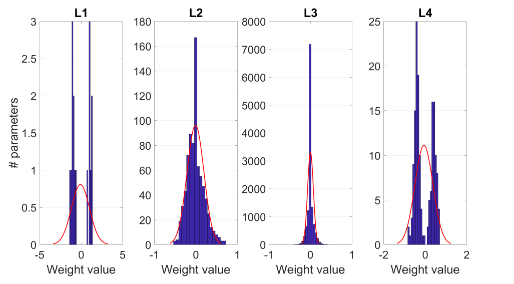
(b) After the 2nd stage, using CU method.



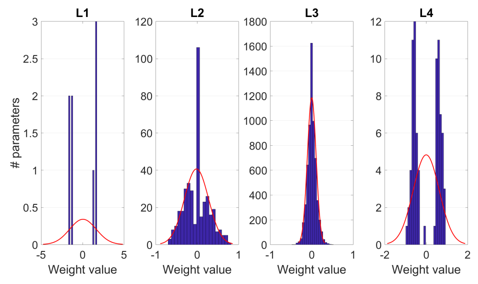
(c) After the 3rd stage, using CU method.



(d) After the 4th stage, using CU method.



(e) After the 5th stage, using CU method.



(f) After the 6th stage, using CU method.

Figure 4.6: Weight distribution of LeNet-5, across different pruning & retraining stages, using the Class-Uniform method.

4.6 Experiments

We apply our methodology not only to LeNet-5 on MNIST dataset, which corresponds to the example provided in Section 4.5, but also on other tasks: LeNet-300-100 on

MNIST (Section 4.6.2), VGG-16 on CIFAR-10 and CIFAR-100 (Section 4.6.3), two other networks, AlexNet and GoogleNet, on CIFAR-100 (Section 4.6.4). A summary of results is reported in Table 4.1, which shows quantitatively the Memory Saving Ratio and the Accuracy Loss due to our proposed methodology. While a shallow Convolutional Neural Network like LeNet-5 can be pruned to achieve the best result in terms of MSR, also deeper networks (VGG-16 and AlexNet) are able to achieve competitive MSR, compared to the other state-of-the-art methods reported in Section 4.2. The complete setup, as well as the simulation environment, is described in Section 4.6.1.

Dataset	**Network**	**AL**	**MSR**
MNIST	LeNet-5	0.11097%	190.75X
MNIST	LeNet-300-100	0.07165%	107.072X
CIFAR-10	VGG-16	-0.2143%	115.382X
CIFAR-100	VGG-16	-0.8324%	91.462X
CIFAR-100	AlexNet	0.0772%	62.727X
CIFAR-100	GoogleNet	0.0772%	15.136X

Table 4.1: Experiment results in terms of Accuracy Loss and Memory Saving Ratio.

4.6.1 HW/SW Setup

We use pyTorch framework [51] for our experiments. We implement sparsity as masked layers, where each mask is multiplied with the weight matrix during the retraining process. The accuracy has been computed by running inference on the validation set, while MSR has been evaluated as the ratio between the nonzero parameters of the baseline model and the nonzero ones of the sparse model. The experiments have been run on Nvidia GTX 1070 GPU, whose specs are reported in Table 4.2. A schematic view of the process flow is reported in Figure 4.7.

4.6.2 LeNet-300-100 on MNIST

LeNet-300-100 is a Neural Network consisting of three fully-connected layers, as described in [32]. We applied our methodology to this Network on MNIST dataset, obtaining a sparsity percentage of 0.93%, which corresponds to a Memory Saving Ratio of 107X. Such result has been achieved after the seventh iteration with constant pruning percentage of 50%. The initial learning rate has been reduced accordingly, from 0.01 of the train-from-scratch process, to 0.003 for retraining.

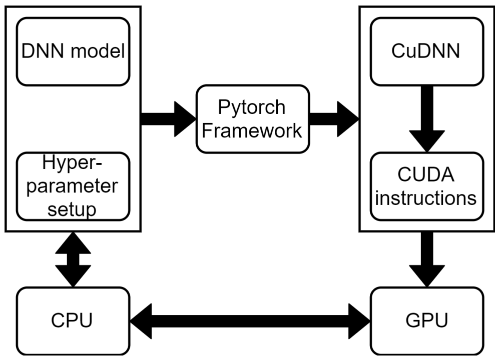


Figure 4.7: SW to HW setup

NVIDIA GTX 1070 specs	
CUDA cores	1920
Memory	8 GB DDR5
Mem. interface width	256-bit
Mem. bandwidth	256 GB/s
Single precision Flops	6.5 TeraFLOPS
Power requirement	150 W

Table 4.2: GPU specs

4.6.3 VGG-16 on CIFAR-10 and CIFAR-100

VGG-16 network is a quite deep neural network, with 13 convolutional layers and 3 fully-connected ones. The model used to be trained on ImageNet dataset is described by Simonyan and Zisserman [42]. We applied some minor modifications to adapt it for other datasets: CIFAR-10 and CIFAR-100. Both datasets are composed of 50000 training images and 10000 test images. While CIFAR-10 contains 10 different classes of images, CIFAR-100 has 100 classes. We applied our methodology on this network and the two datasets provide different results: we obtained a sparsity percentage of 0.87% on CIFAR-10 and 1.09% on CIFAR-100; the respective Memory Saving Ratios are 115X and 91X. For both configurations, the starting learning rate has been set to 0.003 and the pruning percentage to 50%.

4.6.4 AlexNet and GoogleNet on CIFAR-100

Again, CIFAR-100 dataset has been used for training other two DNNs, AlexNet [31] and GoogleNet [44]. Since in their respective original papers, the two networks were designed to be trained on input images of size 224x224, they have been adapted to the size of CIFAR-100 images (32x32). For AlexNet we achieved a MSR equal to 63X, while for GoogleNet 15X. Note that the MSR for GoogleNet is quite low compared to the other experiments, because this DNN is already more sparse than the others in the original form. For this reason, we are not able to compress much the network without affecting the accuracy. The starting learning rate used for AlexNet is 0.003, while the respective value for GoogleNet is 0.03. The pruning percentage set for both processes is 50%.

4.7 Conclusions

In this work we present a simple but at the same time effective methodology, based on Class-Blind pruning scheme, to compress wide and dense Neural Networks (GoogleNet is considered a less dense one, as explained in Section 4.6.4) from 60X to 190X, without affecting the accuracy. This result allows to reduce memory and computational requirements, making inference more feasible to deploy on mobile applications. Since our "PruNet" method is orthogonal with other compression techniques, like quantization and weight sharing, the memory savings can be improved further.

Chapter 5

Conclusions

Deep Neural Networks have revolutionized many Artificial Intelligence applications in recent years. They have demonstrated to have great success, thanks to their high level of accuracy that they can achieve. However, high-accuracy DNNs are difficult (if not impossible) to deploy in mobile/embedded systems, which have strict resource and power constraints. This challenge can be addressed by applying optimizations to improve DNN efficiency. Optimizations can be applied on two aspects: improve the accuracy and reducing memory requirements.
However, despite the success of our methodologies, accuracy improvements and compression ratios are bounded and presents a limit that cannot be overcome. On the other side, the computation requirements are growing fast as well. Guidelines to design accurate and efficient DNNs from scratch are very difficult to generalize. Moreover, incorporating also the training process in mobile devices would potentially further improve the effectiveness of DNNs, because real-time training (and retraining) allows application-specific and reconfigurable system solutions. Such scenarios must be supported by efficient hardware architectures, that exploits parallelism, real-time and resource constraints.

Bibliography

[1] S. Anwar, K. Hwang, and W. Sung. Structured pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.08571*, 2015.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Soravit Changpinyo, Mark Sandler, and Andrey Zhmoginov. The power of sparsity in convolutional neural networks. In *ICLR*, 2017.

[4] Justin Chen. Combinatorially generated piecewise activation functions. *arXiv preprint arXiv:1605.05216.* 2016.

[5] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015.

[6] Djork-Arn Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). In *5th International Conference on Learning Representations*, 2015.

[7] Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *JMLR*, 12:24932537, 2011.

[8] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or 1. *arXiv preprint arXiv:1602.02830*, 2016.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

[10] Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[11] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, 2014.

[12] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon, *arXiv preprint arXiv:1705.07565*, 2017.

[13] Marco Federici, Karen Ullrich, and Max Welling. Improved Bayesian Compression. In *NIPS*, 2017.

[14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Networks. In *AISTATS*, pp. 315323, 2011.

[15] Alex Graves and Jurgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602610, 2005.

[16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *CoRR*, 2015.

[18] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2015.

[19] Song Han, Jeff Pool, John Tran, and William J Dally, Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

[20] Mark Harmon and Diego Klabjan. Activation Ensembles for Deep Neural Networks. *arXiv e-prints arXiv:1702.07790.* 2017.

[21] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1993.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pp. 10261034, 2015.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[24] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017.

[25] Dan Hendrycks and Kevin Gimpel. Generalizing and improving weight initialization. *arXiv preprint arXiv:1607.02488.* 2016.

[26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS*, 2015.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[28] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *NIPS*, 2014.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: An open source convolutional architecture for fast feature embedding. *http://caffe.berkeleyvision.org/*, 2013.

[30] Gnter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. (2017). Self-Normalizing Neural Networks. In *Advances in Neural Information Processing Systems* (NIPS).

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (NIPS), pages 10971105, 2012.

[32] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[33] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[34] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In *ICLR*, 2016.

[35] Darryl D Lin, Sachin S Talathi, and V Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. *arXiv preprint arXiv:1511.06393*, 2015.

[36] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *NIPS*, 2017.

[37] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing.* Citeseer, 2013.

[38] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*, 2017.

[39] Vinod Nair and Geoffrey Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, pp. 807814, 2010.

[40] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. In *CoRR*, 2017.

[41] Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. In *CoNLL*, 2016.

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[43] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overtting. In *Journal of MachineLearning Research*, 2014.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[45] Karen Ullrich, Edward Meeds, and Max Welling. Soft Weight-Sharing for Neural Network Compression. *ICLR*, 2017.

[46] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *NIPS*, 2011.

[47] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. *arXiv preprint arXiv:1610.00324*, 2016.

[48] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In Advances In *NIPS*, 2016.

[49] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: scaling up image recognition. *arXiv preprint, arXiv: 1501.02876*, 2015.

[50] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[51] pyTorch framework: https://github.com/pytorch/pytorch