POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Optimization and compression methods for recurrent neural networks on-chip

Relatore:
Andrea CALIMERA

Correlatore:
Valerio TENACE

Laureando:
Federico PUGLIESE

October 2017

Table of contents

Acknowledgments

For this thesis, I would like to thank both Prof. Andrea Calimera and Valerio Tenace. They believed in me and in my ideas, giving me plenty of freedom to realize them.

For these five years, I must thank Chiara. She was my lighthouse when the storm came. Also, she taught me to be more realistic and more rational, which was a fundamental aid to achieve this goal. She is still my challenge and my certainty.

I would like to thank my parents, too. Despite all differences, they made all of this possible. They also gave me the greatest of gift: they taught me curiosity.

I must thank my brother. He is my greater supporter behind the curtain. He shares my interests and can stand my long talks about any kind of idea. He is my constant.

I have to thank Simone. He was my mate in this long journey, supporting me during projects, exams and all types of challenges. He made my fussiness feel less alone.

I must thank Federica. She believed in me since the first moment, despite all my flaws. She continuously reminds me the importance of my passions, here and elsewhere.

I have to thank Glory, too. She was always there to listen. She helped my mind during these years, extracting words I did not know to have. I owe her patience and time.

I would like to thank Davide C., Andrea, Davide L., Matteo and Gabriele for their effort in bearing my imagination.

Abstract

Model compression and operations savings are fundamental when dealing with limited-resources devices or computational-intensive applications. Neural networks models are often huge because of their great number of weights, which makes them difficult to be implemented on embedded systems, but also expensive on analytics servers: a reduction is necessary when virtually infinite memory is not an option or power consumption is relevant. While networks are theoretically evolving by a mathematical point of view, with increasing complexity and expressiveness, an eye on the engineering aspect must be kept to make them viable and affordable in critical contexts. However, this kind of model reduction can also have significant effects on mathematical design: results showing a comparable or even higher accuracy could suggest that models must be rethought, for example because of great redundancy and unnecessary complexity.

This is the case of this work. New training and optimization methods are proposed to give the model a proper shape for compression, without lowering its accuracy; actually, in some cases, they can also improve it. Then, a powerful encoding scheme and a smart operations scheduling are introduced to save memory, power and time.

Four fundamental results have been achieved: the chosen LSTM network can be compressed obtaining a $50 - 100$ times smaller model, or even $1000\times$ with a little loss, both with a new optimization method - a projection on a constrained solution space - and with the associated bit-encoding technique; the proposed operations scheduling saves 99.9% of multiplications in matrix-vector products, and 100% in some cases, avoiding the need of a multiplier; cutting away around $90\% - 95\%$ of network's weights and applying the proposed method still improves the accuracy with respect to the normally trained model, or at least can keep it stable to the state-of-the-art one; also, when this percentage reaches 99%, all recurrent connections are eliminated while accuracy keeps almost the same, showing that a feed-forward model is enough for the target task with the aid of these techniques.

Furthermore, a general model compression tool has been developed. It applies all the proposed techniques to the target function representing the desired network workflow; finally, it returns the compressed and encoded model.

Chapter 1

Introduction

1.1 Motivation

Models compression allows the implementation of dedicated and compact hardware accelerators for neural networks. This is useful for their employment in critical contexts where resources are limited or where their usage is intensive, like in analytics servers. Dedicated devices can save memory, power and time: compression can significantly lower the huge sizes of these models, but also regularize their structures to perform less mathematical operations, or simpler ones, to obtain the same results.

There are some proposed hardware accelerators in literature and different possible compressions methods [1][2][3], but most of the works concentrate on feed-forward networks, leaving the field of recurrent neural networks less explored. Even if some works try to simplificate recurrent models by a design point of view proposing new schemes and architectures [4], performing optimizations and compressions is still fundamental to reach optimal results during their training. The proposed solutions are quite similar: they use pruning to cut neuronal connections [5] and quantization to approximate the remaining ones [6], but also a binarization of weights values [7] is possible. All these aspects will be better discussed in this work while explaining the new proposed techniques.

1.2 Recurrent neural networks

1.2.1 Neural networks generality

Artificial neural networks are computational models inspired to the brain. They are very complex functions, with many parameters, that can associate an input to a desired output; these parameters work as coefficients for inputs, which are multiplied by them in many different combinations to obtain precise output values.

Neural networks can be used to label some data or in general to elaborate information in order to receive an answer. They are trained to properly do so, and the aim of this training is indeed to find the right values for these parameters to mathematically transform each input in its right output. In this way, intelligence is obtained *simply* through a perfectly

shaped function, exactly as our brain works with its electrical and chemical impulses: however, while the brain works in the *frequency* domain, artificial neural networks work with *intensity*. Instead of having frequent or infrequent electrical impulses to excite or inhibit neuron functionalities, artificial networks will work with positive or negative values to accomplish the same functions.

To understand the workflow of a simple artificial network, we can look at this so-called single layer perceptron model:

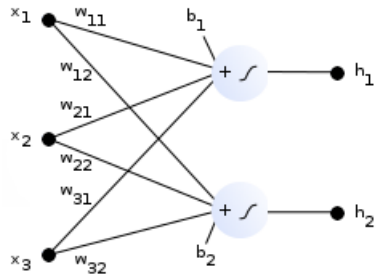
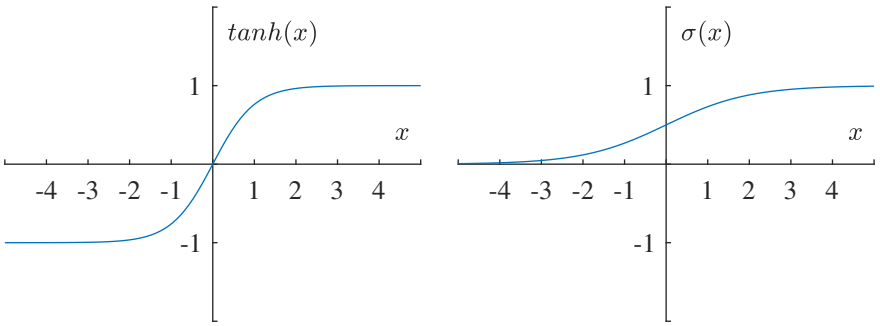


Figure 1.1: Single layer perceptron

Here, in Figure 1.1, three inputs are given to two neurons. Each neuron computes a single output: it multiplies each input value by a different coefficient and adds them up together with a constant offset; then, it uses a particular non-linear function to adapt the result. The non-linear functions relevant for the purposes of this work are only two: the hyperbolic tangent $tanh(x)$ and the squashing or sigmoid function $\sigma(x)$. They are defined in Figure 1.2.



$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ $\sigma(x) = \frac{1}{1+e^{-x}}$

Figure 1.2: Definitions of hyperbolic tangent and sigmoid function.

Therefore, the behavior and the computation performed by a single neuron $k \in \{1, 2\}$

can be written in this way:

$$h_k = tanh(w_{k1}x_1 + w_{k2}x_2 + w_{k3}x_3 + b_k) = tanh(\sum_{j=1}^{3} w_{kj}x_j + b_k)$$

So, if we consider a more compact vector notation, calling $\vec{x} = (x_1, x_2, x_3)^T$ the three-components input and $\vec{y} = (y_1, y_2)^T$ the two-component output, the above formulation can be expressed by a element-wise hyperbolic tangent of a matrix-vector multiplication with a bias vector $\vec{b} = (b_1, b_2)^T$:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{12} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} tanh(y_1) \\ tanh(y_2) \end{pmatrix}$$

which can be written in compact notation as

$$\vec{y} = W\vec{x} + \vec{b}$$

$$\vec{h} = tanh(\vec{y})$$

In this sense, a network is a function of the inputs: it computes $\vec{h}$ from $\vec{x}$ by linking each input element to a neuron through a connection which has a coefficient, called *weight*. The set of all weights becomes the weights matrix W. $\vec{h}(\vec{x})$ is indeed

$$\vec{h} = tanh(W\vec{x} + \vec{b})$$

Other types of networks can have different expressions, but they can be rewritten to get a similar formulation.

However, the strength of neural networks stands in their multi-layer structure: connecting many layers can allow creating complex functions to solve complex tasks. In a fully-connected network, each neuron is linked to all the outputs of the previous layer, for example to create a so-called multi-layer perceptron shown in Figure 1.3. The dimension of each layer can be chosen with proper rules, but once the model is fixed, its chain of computations will produce the desired output only if weights are chosen correctly. This is a non trivial problem for large and deep networks and it is especially hard when their structure becomes less regular than this one.

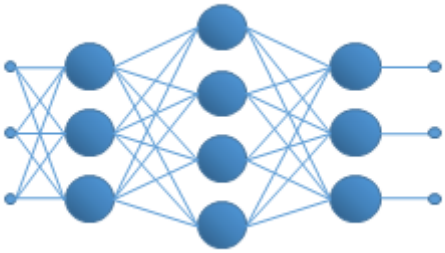


Figure 1.3: Multi-layer perceptron

1.2.2 Long Short-Term Memory (LSTM)

The previous multi-layer perceptron was an example of *feed-forward network*: in this kind of models data flows only in one direction, from the beginning to the end of the structure. However, in some cases, it could be useful to redirect a previous output backwards, for example when dealing with intrinsic sequential data: instead of giving the entire sequence as a single input to a huge network, they can be passed to the network one by one. Furthermore, the length of the sequence could be unknown, so that designing a proper sized network is impossible.

A possible application can be understanding a question and replying with more or less complex phrases, such as reading something about *The Lords of the Rings* and answering to the questions *"Where is Frodo now?"* or *"Where is the ring?"* [8]; it can be looking at an image and giving it a description [9]; it can be predicting the presence of an exoplanet around a star through a temporal series of measures of its brightness to find if an eclipse has occurred [10] or classifing supernovae [11]. The network will understand dependencies between single elements of the sequence by feeding itself with its previous output at each step, which of course will depend on the previous input(s).

For example, a simple *recurrent neural network* can be the one shown in Figure 1.4.

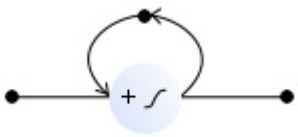


Figure 1.4: Simple recurrent neural network.

Here, in Figure 1.4, all outputs are redirected backwards, if we consider them as vectors. Still, some studies specifies that, due to its structure, this kind of network cannot work well.

This is known as the *exploding or vanishing gradients problem* [12], which has been solved with a new kind of neural network: Long Short-Term Memory (LSTM) [13] [14]. LSTM are based on a different type of neuron, which is far more complex than the perceptron one, and can decide what to remember and what to forget of the past: in this way, it can be very powerful in managing data dependencies. As before, this neuron can be seen in its vector form, in the sense of a layer: in this way, the description will be valid for each possible dimensions of inputs and outputs, which can be also 1 - a single neuron.

LSTM adds three important components to the standard neuron: they are the *input gate*, the *forget gate* and the *output gate* (Figure **??**).

Data enters the network - or neuron - normally: the two contributions of the previous output at step $t-1$ and the current input at step t are weighted and added up. Then, the resulting quantity goes through the hyperbolic tangent.

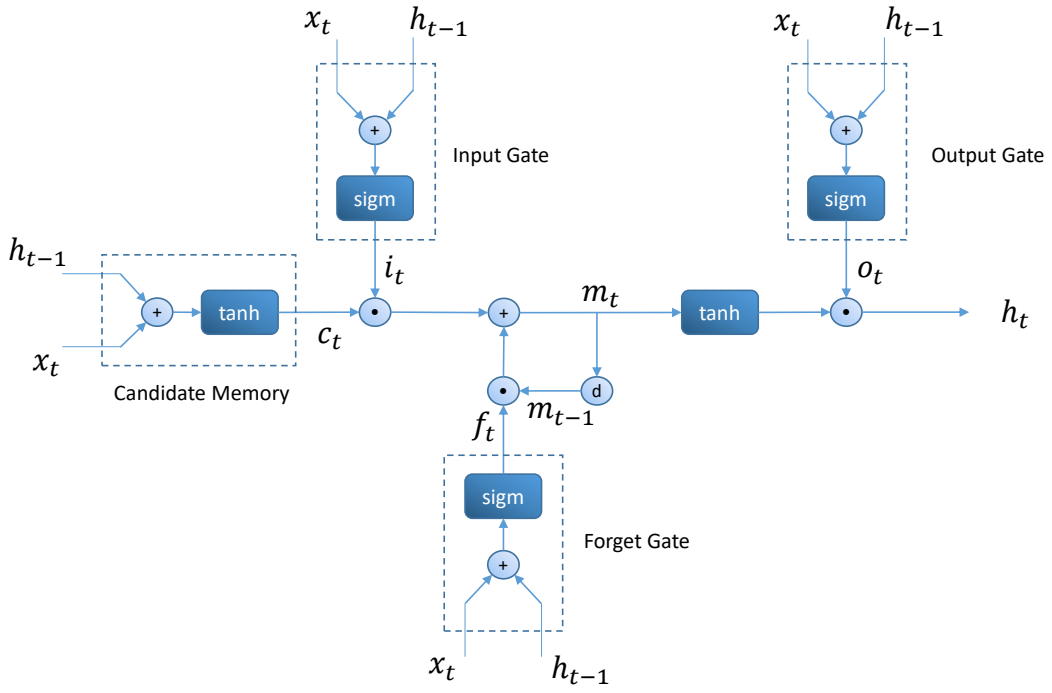


Figure 1.5: LSTM neuron.

Now, differences begin: while normally this value would simple be the output, here it is instead reduced by a number between 0 and 1, because it is multiplied by the result of a sigmoid. Generally speaking, each element of the *candidate memory* $\vec{c}_t$ is multiplied by its respective element in the input gate vector, $\vec{i}_t$. In this sense, the input gate can select

what and how much to remember of the candidate memory - which explains both the vectors names - according to the previous output and the current input. This is obtained by letting pass an element of $\vec{c}_t$ with a value close to 1 in $\vec{i}_t$ or by truncating it with a value close to 0, differently for each of $\vec{c}_t$ elements. If weights are set properly, the network truly learns to do so. This element-wise operation can be written as:

$$\vec{c}_t^{\,*} = \vec{i}_t \odot \vec{c}_t$$

where $\odot$ denotes the Hadamard product, a multiplication between two vectors of the same length, component by component.

This modified $\vec{c}_t$ will contribute to the current memory of the LSTM, $\vec{m}_t$: this value is truly stored by the neurons through all the steps. The old memory $\vec{m}_{t-1}$ is multiplied by the forget gate vector, $\vec{f}$. This works exactly as the input gate: it selects what and how much to remember of the previous memory by multiplying it with values between 0 and 1, using $\vec{h}_{t-1}$ and $\vec{x}_t$ to decide. The modified previous memory is added to $\vec{c}_t^{\,*}$: usually, it is forgotten something that will be replaced by the new information through this sum. This combination of new and old will become the current memory $\vec{m}_t$.

$$\vec{m}_t = \vec{i}_t \odot \vec{c}_t + \vec{f}_t \odot \vec{m}_{t-1}$$

This value is again squashed with a *tanh* to avoid explosions of its magnitude, thus restricting it again between -1 and 1. Then, finally, an output gate decides what and how much to propagate outside the network/neuron. This output will be used by the following layer, but also by the LSTM itself with the recurrent connections. So, the outputs gate vector $\vec{o}_t$ multiplies $\tanh(\vec{m}_t)$ to obtain the current output $\vec{h}_t$, which will soon become the previous output for step $t+1$.

$$\vec{h}_t = \vec{o}_t \odot tanh(\vec{m}_t)$$

It is important to notice that data dependency is very strong: for example, each element of the current input and the previous output contributes to decide what to drop from the memory through the forget gate. Each component of $\vec{f}$ is indeed a weighted sum of every element of $\vec{h}_{t-1}$ and $\vec{x}_t$. So, dimensions - neurons - are not independent: the network is fully-connected like a multilayer perceptron, but with recurrent connections too.

It is possible to express the complete mathematical model of LSTM by introducing three groups of weights matrices for the gates and one group for the candidate memory as

- W_i and U_i, the ones used to multiply $\vec{x}_t$ and $\vec{h}_{t-1}$ respectively in the input gate;

- W_f and U_f, matrices of the forget gate for current input and previous output;
- W_o and U_o, matrices of the output gate;
- W_c and U_c, matrices used for the candidate memory;

also, four bias vectors b_i, b_f, b_o and b_c for each gate and the standard candidate memory will be used.

This model can finally be written as a series of equations:

$$\vec{i_t} = \sigma(W_i\vec{x}_t + U_i\vec{h}_{t-1} + b_i)$$
$$\vec{f_t} = \sigma(W_f\vec{x}_t + U_f\vec{h}_{t-1} + b_f)$$
$$\vec{o_t} = \sigma(W_o\vec{x}_t + U_o\vec{h}_{t-1} + b_o)$$
$$\vec{c_t} = tanh(W_c\vec{x}_t + U_c\vec{h}_{t-1} + b_c)$$
$$\vec{m}_t = \vec{i_t} \odot \vec{c_t} + \vec{f_t} \odot \vec{m}_{t-1}$$
$$\vec{h_t} = \vec{o_t} \odot tanh(\vec{m}_t)$$

1.3 Training, validation and test set

All these three sets of data are divided into inputs and outputs. They express the desired behavior of the network: given that specific input, the network must answer with the relative specific output. If not, the training algorithm will try to modify its weights in order to achieve this goal.

This algorithm will work only with the training set which, like the other two, has a bunch of input-output couples. The assumption is simple: if this set is well gathered and general enough, the weights so found will be able to behave in the same way for other similar data. In other words, if the network has been properly trained to provide the right outputs to this general and representative set of inputs for a certain task, the same network will also give right outputs when new similar data will be proposed to it.

In this sense, a test set is useful to understand the performances of the training phase. It has the test values to propose to the network: they will provide a measure of its accuracy and ability to fulfill its task. This set will not be used to modify the weights, but it is just a performance indicator for experiments and tunings of the training algorithm, also useful to have a standard to compare results between different architectures and works.

In order to build a bridge between these two sets, the validation one is introduced by picking a little number of examples from the training set. Like the test set, it will not be used to modify the weights, but it will contribute in some ways during the training phase:

it will try to avoid the known problem called *overfitting*. Overfitting happens when the training algorithm modifies the weights to perfectly provide the training-relative outputs in a maniacal manner: if this happens, the network did not learn to generalize over the the training examples but instead to simply replicate its outputs. Therefore, it could not be able to provide the right answers to similar but not equal inputs. This problem usually happens when the error on the training set is very low, almost zero; to understand if it truly is a critical situation, measure the error on validation data, like a test set, can be useful. If this error is also low, then we can conclude that the network did learn to generalize; but if it is high, overfitting has happened. So, it is useless to pick the best combination of weights with respect to the training error only: instead, a smarter choice is to pick the model with the lowest validation error. In this way, the network will continue to normally learn with the training set, but picking this model is a way to be quite sure that a good test error will be reached, because the validation set simulates the function of a test set. A typical evolution of these errors during the training phase is indeed the one shown in Figure 1.6.

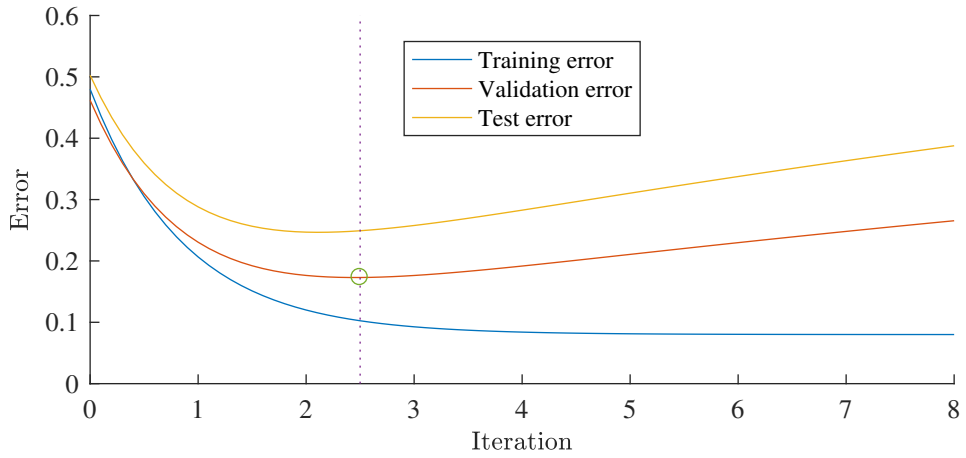


Figure 1.6: Errors during training steps. The best model is the one with minimum validation error.

The validation set size must be small, because its elements are taken away from the training set: the network will have less examples to learn from.

1.4 Experimental setup

To test the proposed methods, a light network was chosen. Its aim and architecture is not important for the study: all these techniques are thought without looking at tasks or

models; however, a quickly trainable recurrent network had to be picked to perform the experiments.

The chosen network is the one proposed by [15] [16]. It is used for *sentiment analysis*, which in this case is a task that consists in stating if a certain review is positive or negative. Even if this task can seem easy, sarcasm and cultural references are something that only we, humans, can easily understand thanks to our previous experiences; and sometimes we too fail in doing so. An artificial model must not only understand the language, but also all its hidden shades that we give for implicit.

Each word in a dictionary of $S = 10^5$ entries will be mapped to a vector of $D = 128$ float-32 components; then, the review of length K will be given to an LSTM network with float-32 weights, one word-vector at a time. LSTM's K outputs on 128 elements will be averaged on a single 128 components vector and given to a logistic regression layer, which is a classical way to obtain the probabilities of the two classes, positive or negative. In this way, the network can tell the nature of the review: the class with the highest probability will be assigned to it. Figure 1.7 depicts this workflow.

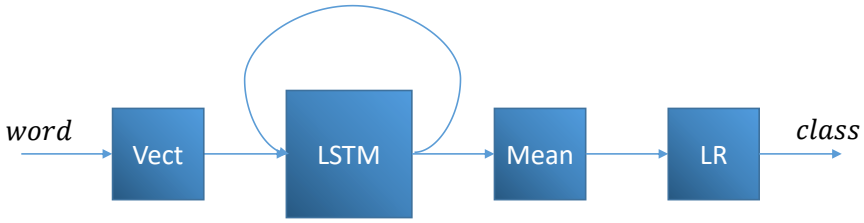


Figure 1.7: Network workflow.

As a side note, the LSTM was also substituted by a GRU [4], one of its variants, in the last phases of this work, to test these techniques and the generality of the developed tool. Similar results were reached, so only the in-depth analysis on LSTM will be reported.

The reviews will be picked from the standard and famous *IMDB large movie reviews dataset* [17]: it has 25000 reviews for training and other 25000 reviews for testing. Both sets are balanced with positive and negative ones.

To lower the amount of time required for training, different subsets of the reviews will be used, both for the training set and the test set. The validation set is built randomly taking the 15% of the training examples.

- Type A - this is a type where the training set is built taking all the shortest reviews, with a length lower or equal to 100 words (2103 reviews). The test set is build taking 500 random reviews with no restrictions;

- Type B - training set: reviews with length lower or equal to 150 words (5847 reviews). Test set: 6000 random reviews, again with no restrictions;
- Type C - training set: reviews with length lower or equal to 300 words (17148 reviews). Test set: 12000 random reviews, again with no restrictions;
- Type D - the entire training and test sets.

All the techniques will work on the weights, leaving the little amount of biases untouched. This is the same approach used by other works, such as [1]. The training is performed end-to-end: the vector representations of words are learned during the training phase and modified during the application of the other techniques. The network will also be trained with *dropout* [18], which is a lateral technique to improve its performances, as the standard implementation of [15] does. However, dropout will not be used during the optimization methods proposed in this work.

1.5 Thesis structure

After this summary of the context, *Chapter 2* will introduce the theoretical background useful to understand the following work and the proposed techniques, explaining the basic concepts of neural networks and their training; it will also describe the model chosen to test these methods.

Chapter 3 will discuss the well known pruning technique and some variations experimented in this work; pruning is indeed the starting point for all the new proposed methods, and its fundamental properties are necessary to obtain good results during the following phases.

Chapter 4 will explain the first new method, called *clustering*. It was born as an attempt to regularize neural networks by distorting their parameters. Its result will be improved by its evolution, introduced in the following chapter; nevertheless, it is a quite good option to reduce the complexity of a network.

Chapter 5 will describe the second new method, and the core of this work, which is called *spiking*. Its mathematical bases are stronger than the ones of clustering, which was a sort of simpler experiment to understand if the kind of domain-transformation proposed by this technique could be effective.

Chapter 6 will provide multiple ways to save memory through network encoding, but also a smart operations scheduling to save multiplications. These are the true motivations that led to the development of clustering and spiking techniques: while the previous

chapters will introduce them in a formal way, chapter 5 will explain and exploit the opportunities they offer.

Chapter 7 will show an high-level overview of the developed general compression tool that implements the previous methods.

Chapter 2

Theoretical background

There are different methods to train a network, which is the process used to find the right values for all its weights and biases. Although evolutionary algorithms have been proposed, the standard procedure uses *iterative methods*, which follow the values of the derivatives of a *loss function* to find its minimum - the lowest error: in fact, the loss function expresses a difference between the desired outputs and the network's ones when a set of training inputs are given to it. This concept will be better explained below.

2.1 Loss function

The loss function $\mathcal{L}$ is the objective to minimize: it expresses the error between the actual output of the network and the desired one. Of course, it depends on the weights, so that it is possible to minimize it with respect to them. Given the set of all the weights, which, for ease of modeling, will be now considered as a vector $\vec{w}$ by concatenating all the rows of the weights matrices and all the biases together, a generic loss function is

$$z = \mathcal{L}(\vec{w}, \vec{x})$$

Given a single input $\vec{x}_k$, it computes the error z_k as a sort of difference between the desired output $\vec{d}_k$ and the current output of the network $\vec{h}_k$ produced by $\vec{x}_k$. An example of this function can be

$$\mathcal{L}(\vec{w}, \vec{x}_k) = \left|\left|\vec{d}_k - \vec{h}_k\right|\right|^2$$

which provides a measure of similarity between the two outputs. It is important to notice that all this kind of functions must have a minimum to reach: in this way, it is possible to apply the derivative-based method mentioned above.

However, we are usually interested in computing the overall error among an entire set; so, we can define the total loss as the sum of all the single errors on the elements of this set. This is possible because errors are always positive, so that minimizing the sum implies minimizing also the single contributions. To avoid explosions of this sum, we can use instead an average to divide it by the number N of the examples in the set $\mathcal{S}$. The

total average error or loss $\mathcal{L}(\vec{w})$ over $\mathcal{S}$ can be expressed as

$$\mathcal{L}(\vec{w}) = \frac{1}{N}\sum_{\mathcal{S}} \mathcal{L}(\vec{w}, \vec{x}_k) = \frac{1}{N}\sum_{\mathcal{S}} \left|\left|\vec{d}_k - \vec{h}_k\right|\right|^2 \tag{2.1}$$

The $\vec{w}$ dependency is not explicit, even if $\vec{h}_k$ depends on it. This will be a problem solved by the introduction of *backpropagation* (section 2.3).

2.2 Stochastic gradient descent (SGD)

Gradient descent is an iterative method to minimize a function. When it is not possible to analytically find a optimal solution in closed form, for example because of the complexity of the expression or because of its huge number of parameters, iterative methods are an useful tool. Gradient descent is guaranteed to find the global minimum only for convex functions; otherwise, the found solution is just a local minimum. This is true if the algorithm does not stop in a *saddle point*, which is a point where derivatives are equal to zero, but this can be avoided with some remarks. However, a local minimum is enough for neural networks: all the solutions found in many experiments since today have proved gradient descent's efficiency. This is a good compromise: networks are too much complex, while this method is reasonably simple and provides very good results anyway.

As the name suggests, this method follows a negative slope of the objective function in order to fall down, until a minimum is reached. It is an iterative exploration of the solution space aimed to find the best point, in this case represented by the optimal $\vec{w}$. This is obtained by observing that if we compute the gradient of a function, which is defined as the vector of its first-order partial derivatives as

$$\vec{\nabla}\mathcal{L} = (\frac{\partial\mathcal{L}}{\partial w_1}, \frac{\partial\mathcal{L}}{\partial w_1}, ..., \frac{\partial\mathcal{L}}{\partial w_N})$$

and we calculate its value in a point $P(v_1, v_2, ..., v_N)$, the direction and verse of $\vec{\nabla}_P\mathcal{L}$ will tell where to move from P to maximize the function $\mathcal{L}$. So, by taking $-\vec{\nabla}_P\mathcal{L}$, we will know symmetrically how to minimize it.

To understand this concept, we can study the function

$$f(x, y) = x^2 + y^2$$

which has a gradient equal to

$$\vec{\nabla} f = (2x, 2y)$$

If we are in $P(20, 0)$, we know that moving according to $-\vec{\nabla}_P f = (-40, 0)$ will take us on a lower value of $f(x, y)$: this is obtained by updating the two variables x and y in the gradient verse, which in this case stands for increasing x, as the normalized versor is negative in x: $\vec{e} = (-1, 0)$. This also means that altering y will not produce any benefits: this is true because the y co-ordinate of P is already the same of the optimum, which is obviously $H(0, 0)$. Figure 2.1 shows it in graphical way, drawing a normalized $\vec{\nabla}_P f$ in orange: it points towards the maximum, which in this case tends towards infinity; it also shows $-\vec{\nabla}_P f$, which has the same direction of the previous vector but opposite verse. Indeed, $-\vec{\nabla}_P f$ would make P fall down towards H.

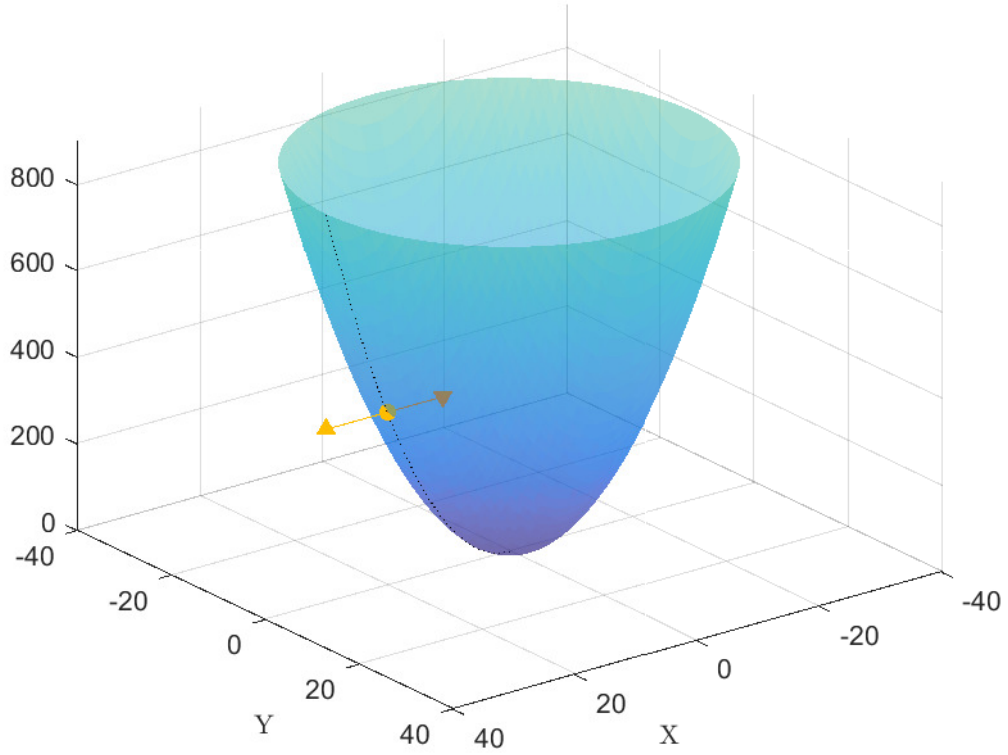


Figure 2.1: Gradient $\vec{\nabla}_P f$, in orange, and its opposite towards the minimum.

However, derivatives are significant only in the neighborhood of P with radius r, which is $N_r(P)$, only if $r \to 0$. So, in the case of $\mathcal{L}(\vec{w})$, if we want to update the current point $\vec{w}_t$ with the information of the gradient, we must take a very small $\Delta\vec{w}$:

$$\vec{w}_{t+1} = \vec{w}_t + \Delta\vec{w}$$

in this way, w_{t+1} will be close enough to w_t and, as $\Delta\vec{w}$ goes according to $-\vec{\nabla}_P\mathcal{L}$, the $\mathcal{L}$ will be *decreasing* in this direction. Therefore,

$$\mathcal{L}(\vec{w}_{t+1}) < \mathcal{L}(\vec{w}_t)$$

which is the desired behavior. We can update the point with this formula:

$$\vec{w}_{t+1} = \vec{w}_t - \eta\vec{\nabla}\mathcal{L}(\vec{w}_t)$$

which moves $\vec{w}_t$ on the same verse of $-\eta\vec{\nabla}\mathcal{L}(\vec{w}_t)$, thus minimizing $\mathcal{L}$. This basic gradient descent version performs this update at each iteration, slowly; it multiplies the gradient by a small factor η which is called *step size*, or *learning rate* when dealing with neural networks. If η is chosen properly and decreases over time, a good solution can be found.

However, computing $\vec{\nabla}\mathcal{L}(\vec{w}_t)$ can be very expensive. Looking at the definition of $\mathcal{L}$ in equation 2.1, the gradient must be calculated as:

$$\vec{\nabla}\mathcal{L}(\vec{w}_t) = \frac{1}{N}\sum_{\mathcal{S}}\vec{\nabla}\mathcal{L}(\vec{w}_t, \vec{x}_k) \tag{2.2}$$

so, the sum of all the gradients over the training set. Usually, to have good network performances and generalization, this set is huge: so, a single iteration would be very expensive - and always many iterations are necessary to find a minimum. To overcome this obstacle, mini-batch Stochastic Gradient Descent (mini-batch SGD) was introduced. Instead of computing the sum over the entire $\mathcal{S}$, each iteration takes M random examples of the set and approximates the gradient just with these M contributions: in other words, a subset $\mathcal{S}_M$, called *mini-batch*, is taken at each step.

$$\vec{\nabla}\mathcal{L}_M(\vec{w}_t) = \frac{1}{M}\sum_{\mathcal{S}_M}\vec{\nabla}\mathcal{L}(\vec{w}_t, \vec{x}_k)$$

So, the update formula becomes:

$$\vec{w}_{t+1} = \vec{w}_t - \eta\vec{\nabla}\mathcal{L}_M(\vec{w}_t)$$

Each iteration will take a different mini-batch, until every example of the original set $\mathcal{S}$ has been visited once. Then, $\mathcal{S}$ will be shuffled and again M random elements of $\mathcal{S}$ will be taken at each iteration, repeating the process. This shuffling prevents any possible collateral effect of providing the examples in a specific order to the network.

Mini-batch stochastic gradient descent has also another positive aspect: it can allow

the search to escape from saddle points or worse local minima, where $\vec{\nabla}\mathcal{L}(\vec{w}_t) = \vec{0}$. The stochasticity obtained by picking random examples can differ the approximated gradient $\vec{\nabla}\mathcal{L}_M(\vec{w}_t)$ from $\vec{\nabla}\mathcal{L}(\vec{w}_t)$: so, these little stochastic fluctuations will allow the point to escape from these bad situations to possibly reach a better local minimum.

Mini-batch SGD is an intermediate version between the proper gradient descent and the standard stochastic gradient descent: the last one, SGD, approximate the total gradient with a single example instead of M. Of course, this leads to slower convergence, as a single contribution is not enough to estimate the true $\vec{\nabla}\mathcal{L}$ in a reasonable way.

2.2.1 AdaDelta

As choosing the proper learning rate η is fundamental for results, many formulations and variations have been proposed. *AdaGrad*, *RMSProp*, *Adam* are just some examples, as explained by [19]. However, *AdaDelta* is one of the common ones and it will be used in this work for its good performances and properties.

Instead of having a unique and arbitrary η, *AdaDelta* proposes an ad-hoc adaptive learning rate: ad-hoc because it is customized for each weight; adaptive because it changes over time by itself, thus avoiding the necessity to manually tune its evolution before SGD algorithm runs. *AdaDelta* extends *AdaGrad*, which stands for *adaptive gradient*. *AdaGrad* divides η by a contribution which depends on the previous history of gradients, to adapt it and decreasing it over time, which is a very good property as explained before. Calling $\vec{e}_t = \vec{\nabla}\mathcal{L}_M(\vec{w}_t)$ for brevity where $e_{t,k}$ is one of its single elements - so a partial derivative with respect to w_k -, we can introduce this factor:

$$Q_{t,k} = \sum_{\tau=0}^{t} e_{\tau,k}^2 \tag{2.3}$$

which is the sum of all squared gradients components until the current step t. The standard update formula

$$w_{t+1,k} = w_{t,k} + \Delta w_{t,k} \qquad \Rightarrow \qquad w_{t+1,k} = w_{t,k} - \eta e_{t,k}$$

is modified by dividing η by the square root of this factor, which again is intended as element-wise:

$$w_{t+1,k} = w_{t,k} - \frac{\eta}{\sqrt{Q_{t,k}}} e_{t,k}$$

However, this sum is very expensive both in time and memory: so, *RMSProp* introduces an *exponentially decaying average* to accumulate the sum of square gradients in a more

elegant way. This factor will be called EG, average of gradients.

$$w_{t+1,k} = w_{t,k} - \frac{\eta}{\sqrt{EG_{t,k}}} e_{t,k}$$

$$EG_{t,k} = \alpha EG_{t-1,k} + (1-\alpha) e_{t,k}^2$$

where $\alpha < 1$ is a number used for this purpose. *AdaDelta* also uses this decaying average to substitute the simple numeric η: it averages in this sense all the *previous* updates $\Delta\vec{w}$ of the $w_{t+1,k} = w_{t,k} + \Delta w_{t,k}$ formula, so only until $t-1$ because of course $\Delta w_{t,k}$ is the value to be computed:

$$w_{t+1,k} = w_{t,k} - \frac{\sqrt{EW_{t-1,k}}}{\sqrt{EG_{t,k}}} e_{t,k}$$

$$EW_{t-1,k} = \phi EW_{t-2,k} + (1-\phi) \Delta w_{t-1,k}^2$$

So, $\Delta w_{t,k}$ is inevitably and recursively found as

$$\Delta w_{t,k} = -\frac{\sqrt{EW_{t-1,k}}}{\sqrt{EG_{t,k}}} e_{t,k}$$

In vector and simpler notation, in this work this update formula will be expressed substituting

$$\eta_{t,k} = \frac{\sqrt{EW_{t-1,k}}}{\sqrt{EG_{t,k}}} \qquad \Rightarrow \qquad \Delta w_{t,k} = -\eta_{t,k} e_{t,k}$$

and so, using the Hadamard product,

$$\vec{w}_{t+1} = \vec{w}_t - \vec{\eta}_t \odot \vec{\nabla} \mathcal{L}_M(\vec{w}_t)$$

In this way, *AdaDelta* adapts the η in oder to explore directions in different ways, according to their gradients' history.

2.3 Backpropagation through time

Computing the derivatives may seem a problem when $\vec{w}$ dependency in $\mathcal{L}_M(\vec{w}_t)$ is not explicit. Also, the complexity of the layers architecture and their formulas can be overwhelming. However, a simple mathematical concept can break this fake issue: the chain rule.

When we have for example two feed-forward layers, we can represent them as two functions f and g. g's output, which is our $\vec{h}$, is of course connected to the loss function

that we want to minimize by computing its gradients (Figure 2.2).

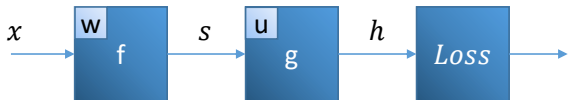


Figure 2.2: Example network with two layers f and g.

However, as each gradient is computed one example at a time thanks to equation (2.2), the single input $\vec{x}_j$ of the set $\mathcal{S}$ is considered as a constant: even if the sub-network f would have $\vec{w}$ as fixed parameters and $\vec{x}$ as variable during its final deployment, the gradient descent will try to search the best $\vec{w}$ for f having $\vec{x}$ as coefficient. In this sense we can state that, during the training phase, f will be a function of $\vec{w}$ having the single $\vec{x}_j$ as fixed parameter. In the same way, g will remain a function of f's output $\vec{s}$ but will become also a function of $\vec{u}$, which is its set of weights (Figure 2.3).

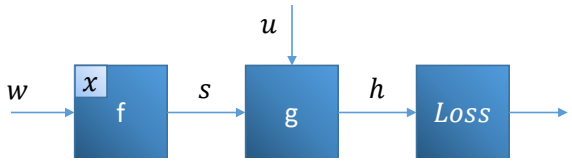


Figure 2.3: Layers written as functions of the weights.

To simplify again, we will consider each output as a scalar, so s and h. The vector generalization can be later found by induction.

Therefore, if for example we want to compute the derivatives of $\mathcal{L}$ with respect to a single weight of f, w_k, we can state that:

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial h}\frac{\partial h}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial h}\frac{\partial h}{\partial s}\frac{\partial s}{\partial w_k}$$

which is the simple repeated application of the chain rule of derivatives: $\mathcal{L}$ is a function of h which is in turn a function of s which is finally a function of w_k. The three derivatives are direct and can be calculated easily. By repeating this process for each weight and each components of vectors it is possible to compute the complete gradient without problems. This process is called *backpropagation*.

For the sake of completeness, the chain rule for vector functions is the following, taking

a composite function $z(\vec{s}(\vec{w}))$:

$$\frac{\partial z}{\partial w_k} = \frac{\partial z}{\partial (s_1, ..., s_N)} \frac{\partial (s_1, ..., s_N)}{\partial w_k} = \sum_{j=1}^{N} \frac{\partial z}{\partial s_j} \frac{\partial s_j}{\partial w_k}$$

When the network is not feed-forward but recurrent, another problem seems to arise. However, again an easy solution can help: if the network is unfolded in time, it is similar to an only forward network with no recursive connections (Figure 2.4).

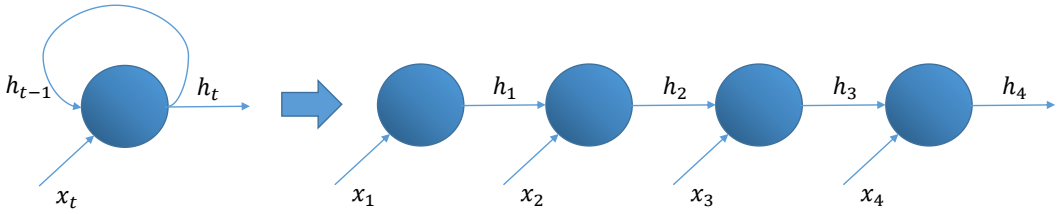


Figure 2.4: Unfolding in time.

In this way, the backpropagation can be applied normally, taking the name of *back-propagation through time*.

Chapter 3

Pruning

3.1 Motivation

The first technique used also in literature either to simplify weights or to find a better network is the so-called pruning. As its name may suggest, it is based on the idea of removing some neural connections by setting their weights to zero. This concept is inspired by different motivations, either philosophical or pragmatic ones.

As replication of biological solutions can be a good starting point when we deal with artificial intelligence, the first reason can be found into the emulation of the mechanism that controls and modify all neural connections in the human brain, especially during its growth. Both artificial and natural neural networks are redundant and error-resilient: they can work if damaged, even if some connections are cut off. As shown in many other works, it is possible to simplify the model so that the accuracy keeps stable [1] [5] [6].

The second motivation is strictly related to the neural network model. Each neuron is fully connected to all inputs, and therefore some connections can be useless: it is possible that a specific input does not contribute (or does not contribute in a sensible way) to a particular neuron functionality. In a worse case, a connection can even alter a neuron work, for example because its weight is far from its optimal value, getting the model stuck in a worse local minimum where all other neurons have learned to compensate this error in a best-effort but not sufficient way. In this sense, pruning can be seen as a sort of jump throughout the weights space, which can be useful to improve the ability of the gradient descent method to escape from a local optimum. It also acts as an extreme form of regularization: it reduces the number of parameters in order to keep the model far from the risk of overfitting.

It is important to notice that this kind of algorithms is normally used to find the optimal network for a specific task, and not to train it. Conceptually, a zero-weight connection is different from an absent one: a zero value is what the training phase found to be optimal for that connection in the target network, but this implies that the target network had to be chosen previously. In other words, choosing the structure of a network - units and links, fully-connected, recurrent - is different from choosing the values in it.

However, in this work, the aim of this technique is not to find an optimal shape for the network: it is to compress an already existent and valid one, keeping the same accuracy or selecting an accuracy-dimension compromise. In this sense, this distinction practically fades: the network will be manipulated in many ways, and in this first phase some links will be discarded regardless from the initial structure, in order to fit a compact compression strategy.

Nevertheless, removing just some connections is not enough. In order to effectively reduce the dimension of the model, the pruning procedure must eliminate a huge amount of weights; however, doing it abruptly will produce a mutilated network with a far worse accuracy: the remaining weights will behave as normal, counting on the work of the other ones [5]. The procedure needs a way to fix the network, working on the connections it still has.

This sentence hides also another fundamental problem: how to select the right weights to prune? What are the essential weights, if they exist?

3.2 Prunable weights selection

Different methods were developed in literature to choose the prunable weights. Some of them are more complex than the other ones in terms of computation; some of them are more widely used than others. For example, Optimal Brain Surgeon (OBS) [20] and Optimal Brain Damage (OBD) [21] are two similar and famous algorithms that try to understand the importance of a weight using a deeper level of information: while normally every decision is made using just the first-order derivatives of the loss function, already computed for training, these two methods use also the second-order ones. As they are *partial*, there exist a derivative for each combination of two variables, thus meaning a quadratic number of operations and values - the Hessian matrix. By gathering all this information, these algorithms choose the weights to prune using the Taylor series of the loss function and an index that explains the importance of each weight. They also repair the network after the cut, but of course their application is expensive, particularly for large network.

Still, another simpler approach is possible. It is based on the idea of using just the information given by the training phase, selecting and pruning all the weights with an absolute value under a certain threshold. Here the name: magnitude-based pruning.

3.2.1 Magnitude-based pruning

Numerically speaking, each weight is a coefficient which multiplies an input in a matrix-vector manner, and so a small weight makes a specific input quite irrelevant in the total sum; this behavior can be approximated by eliminating that input contribution from the computation. In other words, small weights can be approximated by zero-value ones.

This idea finds its effectiveness in the objective of training. Generally speaking, the training algorithm sets a weight high - in absolute sense - if that connection must have a strong excitatory or inhibitory effect, and so a significant influence. In this way it is possible to select the prunable weights knowing that the smallest ones are the best ones to choose, without other computations. As a disadvantage, it requires a little retrain afterwards, to adapt the remaining weights, which will be explained in section 3.3.

Anyway, it was the approach selected by many other similar works, and it worked well also in this one (see section 3.4): it often improved accuracy even beyond any expectations, as the aim of this work was just compressing while keeping accuracy as stable as possible, thus there were no doubts about which approach to select.

3.2.2 Experimental and discarded approaches

Anyhow, before choosing this method, several new alternatives were tried to avoid the use of a technique after the training phase: they were thought to mix it with the pruning phase.

During the training phase, weights move according to the gradient. The smaller the gradient, the smaller the step: and if it keeps small for many updates for a specific weight w^*, it means that w^*'s influence on the loss function is also small. This also means that moving on w^*'s direction is quite useless. Therefore, under certain circumstances, for example if this is true quite everywhere on the solution space, it could be possible to eliminate w^*. This idea need to be of course enforced, to be sure that setting $w^* = 0$ does not have a catastrophic effect on the loss or that it can be compensated by changing some other weights. For example, after the cut, all learning rates could be set to zero, to simulate the restarting of training and to let the other weights have a better and faster adaptation.

Experimental results show that even in its embryonal state, this idea could lead to quite good results, even if not astonishing. However, like for other ideas, it was soon discarded as the common approach with magnitude-based pruning was better studied, already widely used and effective. It also leads to a good point to start with the new

technique proposed in this work, which are the true unstudied and innovative elements of it.

3.3 Pruning time and modality

3.3.1 Time

Being most general as possible, the application of this pruning technique can happen in three moments in time:

- before the training phase;
- during the training phase;
- after the training phase.

3.3.1.1 Before the training phase

Pruning before the training phase means to randomly select some connections and drop them or, better, choose to use a less powerful model, eliminating some links according to a certain criterion. This last approach is equivalent to use a different kind of network and, of course, there is the possibility to prune out fundamental connections if not done carefully. But carefulness would require a study for each architecture to develop a custom solution, which cannot be acceptable for a general compression tool. Alternatively, the pruning algorithm could be informed by other computations before training, but again it would be too complex with respect to the results obtained by simpler approaches.

Nevertheless, setting some weights to zero before the training phase can be seen as a form of initialization. As networks are quite always trained with iterative algorithms, initialization plays an important role in the search of the optimum: therefore, as the desired shape of the distribution of weights after pruning is known (Figure 3.2) and confirmed by other works [5], it could be possible to help the training phase by imposing an initialization similar to the final result.

However, imposing an initial distribution cannot always guarantee proximity to optimal solution: without any other information, it could be possible to set a great number of weights in wrong positions of the distribution, thus making vane the effort of setting the initial point close to the supposed final one. Furthermore, performing other computations to gather this kind of information that can simply be found after the training phase - which has to be performed anyway - can be really a waste of resources and time. So, even if this approach has given some potentially good results, it was soon discarded.

3.3.1.2 During the training phase

Looking at the their evolution during normal training (Figure 3.1), weights values start to significantly change after a certain epoch, then they settle to a local minimum and quite stop evolving, as the algorithm was intended to do.

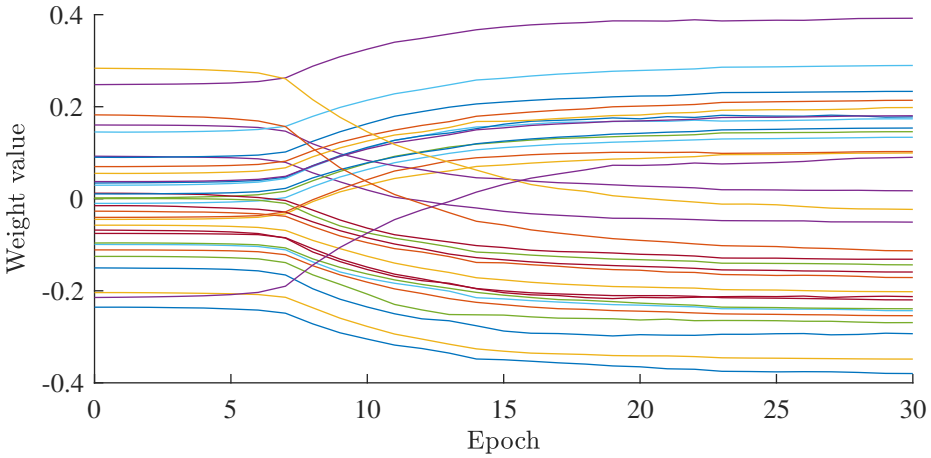


Figure 3.1: Example of weights evolutions, with a different color for each weight.

For this reason, the first idea was to cut the weights between those two moments, or soon after the settlement. This kind of approach offers an interesting possibility: the revocability of pruning decisions. While normally a cut connection is permanently eliminated, here the training could lead to its regrowth. It is a sort of backtrack, where a weight can be restored if the gradient shows strong proofs of improvement and maybe tells what other connection can be cut instead. A similar approach is used for the dropout technique [18] and for the clustering technique introduced in this work (see Chapter 4).

Mixing training with pruning of course increases the time of the training phase itself, as the gradient descent encounters lots of jumps, especially if pruning is performed gradually step by step, which is the way explained and used in subsection 3.3.2. Again, no particular gain has been found by using this approach with respect to the most common one.

3.3.1.3 After the training phase

Several works concerning model compressions use pruning methods after the training phase. As said, other experiments were tried before testing this approach and taking it as a sort of standard. Even if some works suggests the opposite, such as [20], results were still far more than acceptable.

After the training phase ends, the network has reached a local minimum by setting most of the weights to small values (Figure **??**). It is possible to cut those weights, defining what *small* means. Formally, a threshold τ is introduced so that all weights w_j satisfying

$$|w_j| \leq \tau \tag{3.1}$$

are discarded. The right τ must be chosen according to the desired percentage p of pruning: it is the value such that the number of w_j satisfying (3.1) are equal to pN_t, where N_t is the total number of weights in the model.

$$\text{Defining } \mathcal{T}_\tau(w_j) = \begin{cases} 1, & \text{if } |w_j| \leq \tau \\ 0, & \text{otherwise} \end{cases} \text{ it must hold } \sum_{j=0}^{N_t-1} \mathcal{T}_\tau(w_j) = pN_t \tag{3.2}$$

In this work, the right τ is found by dichotomic search (see Chapter 7).

After all the proper weights are eliminated, *i.e.* set to zero, the network needs a brief re-training to adapt all the remaining connections. This new optimization run will start from the pruned network's parameters, as if it was training again the same model but with weights initialized to the ones produced by the cut.

The strength of this approach is in the quick convergence of this phase: typically, a very small number of epochs are necessary to find a new near local minimum [5]. Instead of having ad-hoc formulas to repair the cut, it is the training algorithms to learn how to compensate it by finding a new optimal solution, as it is its job. By constraining the pruned weights to stay at zero, the gradient descent search will navigate throughout the solution space without moving on their corresponding dimensions; alternatively, it is possible to set the corresponding gradient's components to zero.

As reported by several works, this new small training must be done slowing the learning rate in order to stay as much as possible close to the previous optimum and to the starting point - the pruned model right after the cut. This is obtained by introducing into the update formula of the gradient descent algorithm a factor γ_p, whose values might be around 0.0001 [22] (here 0.001 is used):

$$\vec{w}_{t+1} = \vec{w}_t - \gamma_p \vec{\eta} \odot \vec{\nabla} \mathcal{L}_M(\vec{w}_t) \tag{3.3}$$

This kind of slowing down also guarantees a finer search on the solution space, which is reduced in dimensionality, to better explore the neighborhood. This little detail allows maintaining the accuracy and, of course, to maintain also all the decisions taken by the training phase as much as possible, as they should be the best ones.

3.3.2 Step by step approach

However, in order to keep a stable accuracy, finding τ to directly prune the model to the corresponding percentage p is not enough. It is possible to obtain the same accuracy only if pruning is done step by step before reaching the quantity p. Therefore, this percentage must be divided into subsequent p_ks that will gradually prune the model until they reach p_S, where S is the number of steps chosen.

Therefore, the model will be pruned S times, each time with a greater percentage of cut with respect to the initial number of weights N_t. Formally, there will be a sequence of percentage

$$\{p_k\}_{k \leq S} = \{p_1, p_2, p_3, ...p_S\}$$

In this work, this steps will be defined as

$$p_k = k\frac{p_S}{S} \tag{3.4}$$

which are equally separated and gradual steps of pruning, as p_S is divided into S equal parts and p_k refers to the total number of parameters N_t. Indeed, this kind of structure for p_k guarantees that

$$p_k = kp_1 \tag{3.5}$$

This can be easily derived from (3.4) noticing that

$$p_1 = \frac{p_S}{S}$$

Thus, equation (3.5) in turn means that

$$\{p_k\}_{k \leq S} = \{p_1, 2p_1, 3p_1, ...Sp_1\}$$

In this way, each pruning step will prune the same amount of parameters α, which is

$$\alpha = \frac{p_S}{S}N_t,$$

This is true also because each pruning step will keep all the previous $(k-1)\alpha$ pruned weights to zero and will add its α ones.

For this reason, τ will also become τ_S and

$$\{\tau_k\}_{k \leq S} = \{\tau_1, \tau_2, \tau_3, ...\tau_S\}$$

will be a sequence of τ_k such that each of them satisfies (3.2) with respect to the corresponding p_k. In this way, all the S thresholds will correctly follow the weights distribution.

Thinking that equally separated thresholds are associated to equally separated percentages is conceptually wrong, and trying to use them may pose some risks: they would delete more or less weights than desired during itermediate steps, as weights values do not follow the uniform distribution. As shown in Figure 3.2, weights follow a Gaussian distribution. It means that

$$\{\tau_k\}_{k \leq S} \neq \{\tau_1, 2\tau_1, 3\tau_1 ... S\tau_1\}$$

Furthermore, the distribution itself evolves in time during the overall pruning stage, making an a-priori decision on τ_k really dangerous; only searching the right threshold at each step can guarantee the wanted pruning percentage p_k.

Concerning the number of steps S, [22] suggests to prune slowly. Using for example $S = 11$, considering the eleventh and last pruning step p_S close to 100%, it is possible to obtain an overview on the optimal pruning percentages, aside from very good results as shown below (section 3.4).

3.3.3 Universal threshold

When networks have more than one matrix, some considerations can be done about the threshold. These ideas were born in the context of this work and try to study a hidden degree of freedom available when pruning is applied to the model.

Weights can be considered as unique set, without subdividing it among all the matrices: τ will be chosen looking at this simple array of elements. In this way, p weights will be pruned away without caring about the original matrix: if just one of them contains an adequate number of values below the threshold, they will be cut even if they belong exclusively to that single matrix. So, in general, some matrices can be pruned more than the others and just a single threshold is required.

Otherwise, an equally distributed system can be applied, so that each matrix loses the same amount of elements; alternatively, the same *group* of matrices. With this approach, multiple thresholds must be found, separately for each matrix or group of them.

However, the first one seemed the rightful approach. Following the principal idea, weights must be pruned away according to their utility, which is assumed proportional to their magnitude value: if the lowest values are in a single matrix, then that matrix is will be pruned. In this way, it is the algorithm itself to learn which are the best weights to discard, without arbitrary impositions.

Experimental results confirm this idea: even if the recurrent matrices U of the LSTM are pruned in a more aggressive way than W matrices, accuracy keeps stable. When the pruning percentage reaches 99.9%, the entire group of recurrent matrices are set to 0: the method identified them as useless. This outstanding result shows the importance of making the technique as free as possible, because it is strong and powerful enough to learn by itself. All these results will be better discussed below.

3.4 Results

All the choices performed well in this work, from the linearly separated p_k to the number of steps S, showing also a positive effect on reducing the error of the network quite always.

3.4.1 Weights distribution

Figure 3.2 depicts weights evolution during some of the pruning steps.

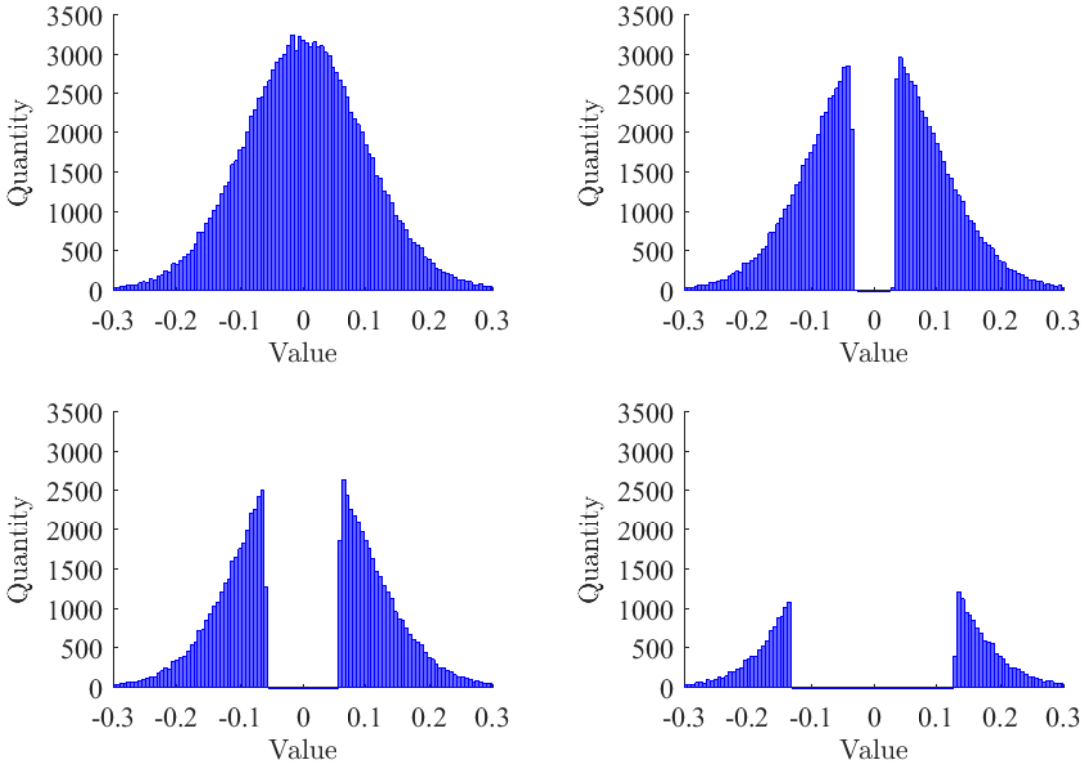


Figure 3.2: Weights distributions. Here, just the ones with 0% (only trained), 27%, 45% and 81% pruning percentages are reported. Pruned weights, which would have produced high peaks on zero, have been removed for readability.

It is easy to notice the particular shape of the distribution after each step: it shows two peaks, as the central core of the Gaussian bell was erased. Also, there is not a sharp separation on the thresholds, as one might expect. This is because the pruning is performed just once at each step, and then all the weights begin to evolve again. Thus, they can be shifted inside the $[-\tau_k; \tau_k]$ region another time. This happens for not so many weights, as the internal slope of the peaks is steeper than the external one, but this possibly allows selecting the next-useless weights and saving other more important ones from the following pruning step: useful weights tends to escape from the possibility of being cut, as the distribution enlarges. This kind of result is obtained also by other works, even for feed-forward networks [5].

3.4.2 Accuracy

To better understand what happens to accuracy during the pruning stage, p_S was again set close to 100%[1] and for type A the number of steps S was doubled to 22, which is also a good value due to the fact that usually the optimal pruning percentage is $90\% - 95\%$, as confirmed by the other works.

Here it is possible to see how this kind of pruning affects the accuracy. The first group of plots describes the accuracy of the chosen network trained with the little dataset of type A, with different seeds. The second group of plots shows the training done with datasets of type B, C and D; for them, S was again halved to 11 due to the long time required for the script to run.

In general, quite always a very high pruning percentage keeps good accuracy or improves it: during the many experiments performed with different seed especially with the little dataset of type A, it was rarely worse.

Furthermore, sometimes a 99.97% pruning percentage shows an interesting loss on accuracy: even if it can be relatively high for type D (4%), it still can be acceptable in some context, when compression is far more important than full precision. Figure ?? show some examples.

However, this high pruning percentage introduces a new, significant result: as U matrices of LSTM contain the lowest values, all of them are pruned away. It means that the recurrent contribution has been erased from the network: the model is now purely forward. Still, it can reach that kind of accuracy. This important result can suggest that a feed-forward network can also work well and it can be enough, maybe if properly designed.

[1] Precisely, its value was set to 99.999%. See Chapter 7 for further details.

Type A shows a great variability (Figure 3.3). Each of them is different in terms of initial training accuracy and evolution during the pruning phase because the portion of validation set and test set were chosen randomly for each seed; also, the regularization effect of pruning is strongly visible here, because the task was easier and the training examples were fewer.

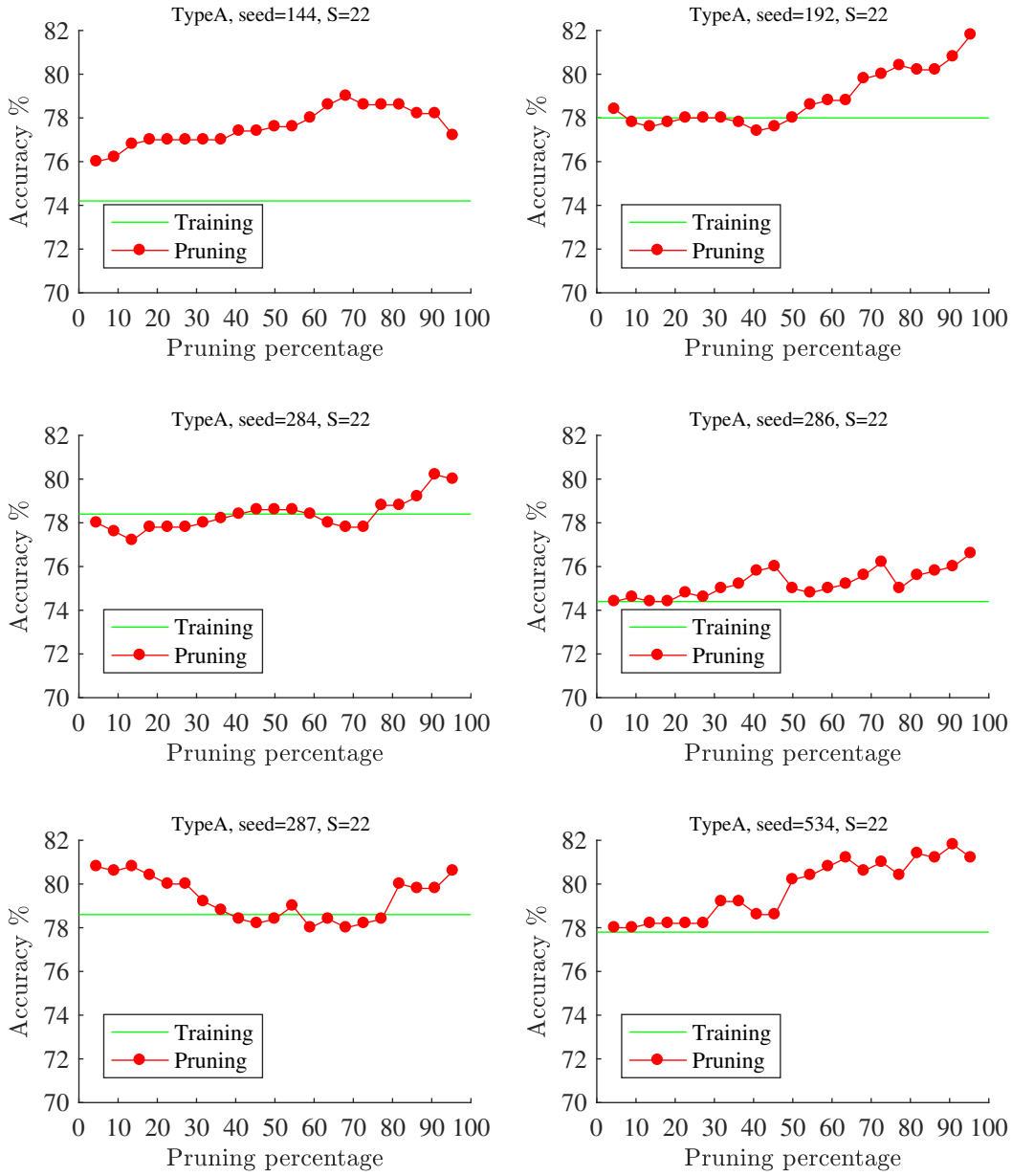


Figure 3.3: Type A, pruning.

In this second group of plots (Figure 3.4), accuracy scale has been set differently for each type for better visualization. Here, accuracy keeps quite stable and its evolutions is always similar for each seed or type.

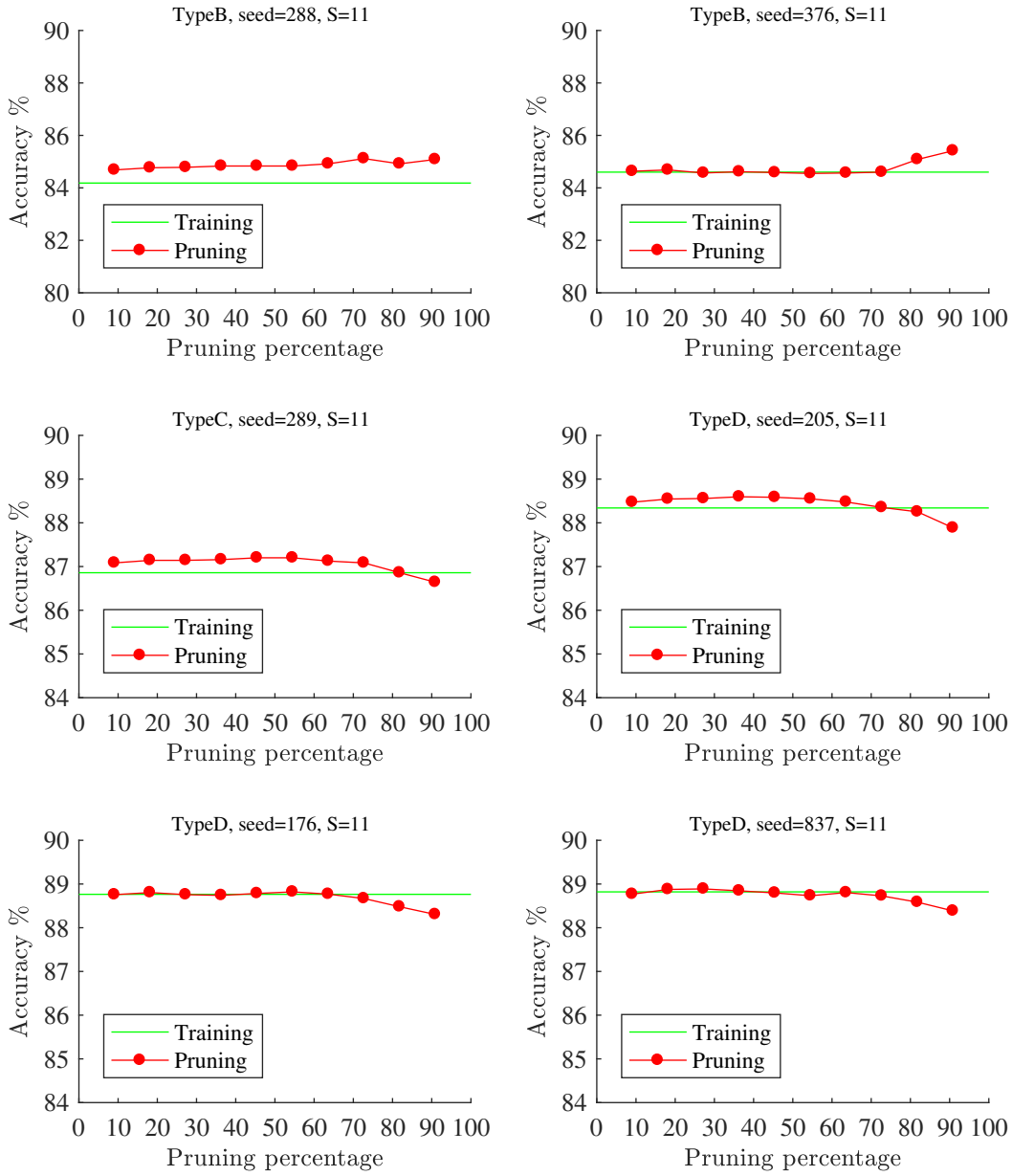


Figure 3.4: Type A, B and C, pruning.

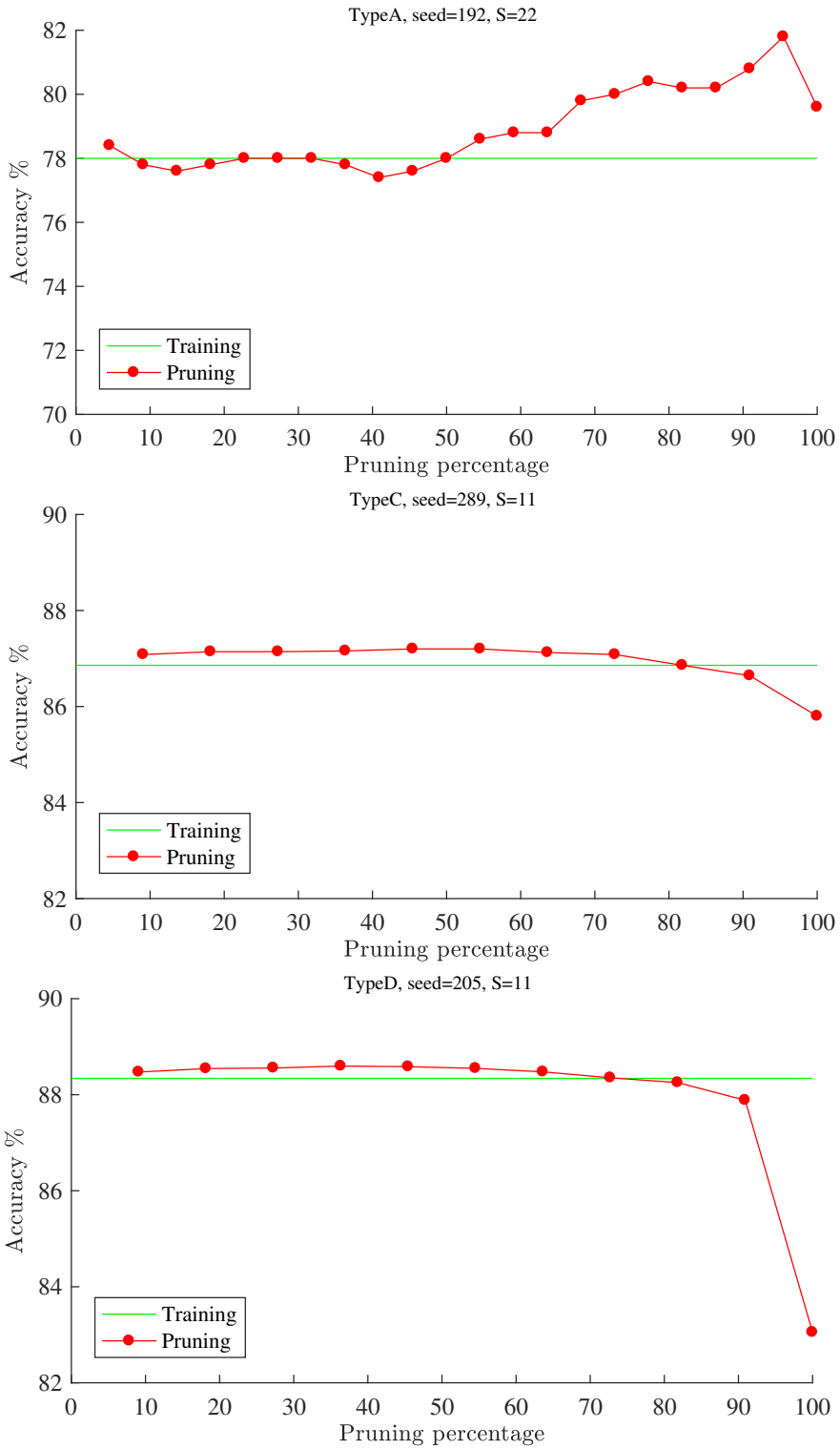


Figure 3.5: Pruning with also 99.97%-step is shown.

Figure 3.6 shows the distribution of the non-pruned weights among the matrices: a blue dot in position (i, j) stands for $w_{ij} \neq 0$. These are the two group of matrices: the four recurrent ones, U, and the four forward ones, W. This is a pruned model of type A with 95% of pruning percentage.

The only matrix with a great amount of non-zero weights is the last one, which is W_c, the matrix of the candidate memory. It also shows regular patterns of non-pruned weights.

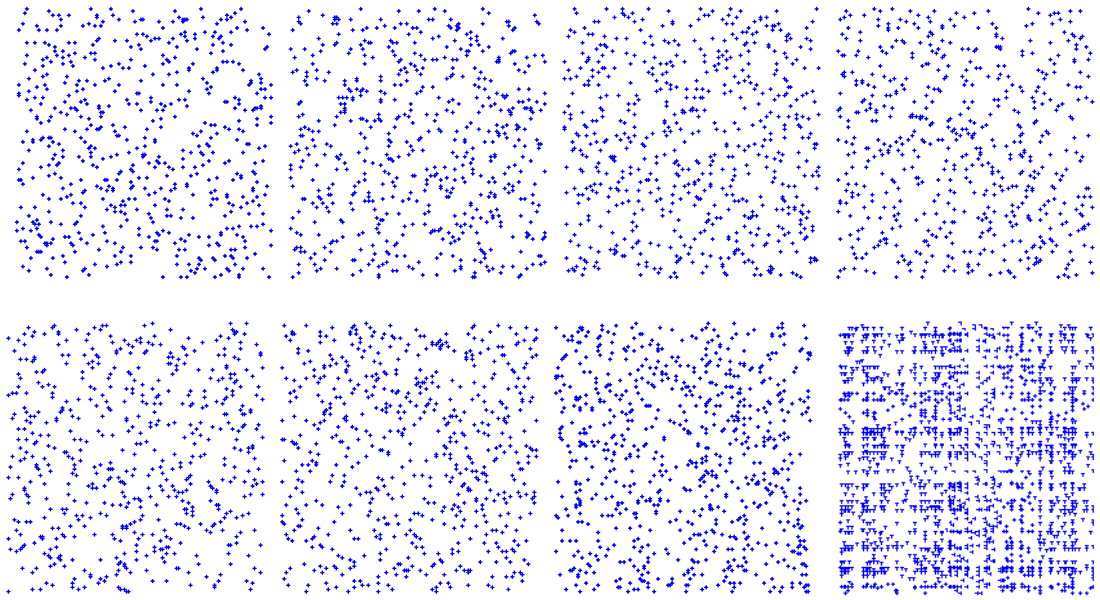


Figure 3.6: Non-pruned weights distribution among LSTM matrices.

When the pruning percentage reaches 99.9%, only W_c survives the pruning stage: again, for example for type D, it shows this kind of regular sparse structure (Figure **??**), which is slightly different from seed to seed.

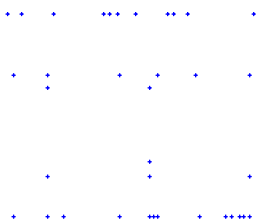


Figure 3.7: W_c is the only survivor with regular structure when $p = 99.9\%$.

This makes sense, because all the forget gate, input gate and output gate are just controllers, while W_c carries the true information from the input. U_c, instead, would

contribute with the old outputs, but, for this kind of task, W_c is apparently enough, like a direct, single and simple neuronal layer.

Chapter 4

Clustering

4.1 Motivation

Looking at the results obtained by the pruning stage, the final distribution of weights suggested a new idea to further compress the network.

Its particular almost-bimodal shape let intend the importance of those values gathered in two symmetrical and short intervals, for the great majority of weights were there. It was clear that the optimization algorithm chose to put them in such a way to minimize the total error, thus underlining the effectiveness of these two ranges of values. Nevertheless, as another work suggests, weights on the edges of the distribution must not be forgotten: following the logic that guided the pruning stage, these high-value weights dominate the computations among the network because their contribution is significant. For this reason, both ranges seemed relevant. This guideline led to the idea of approximating the ranges where the majority of weights were concentrated with two real peaks: all the target weights will be collapsed to a single value, as if it was a clustering center - here the chosen name.

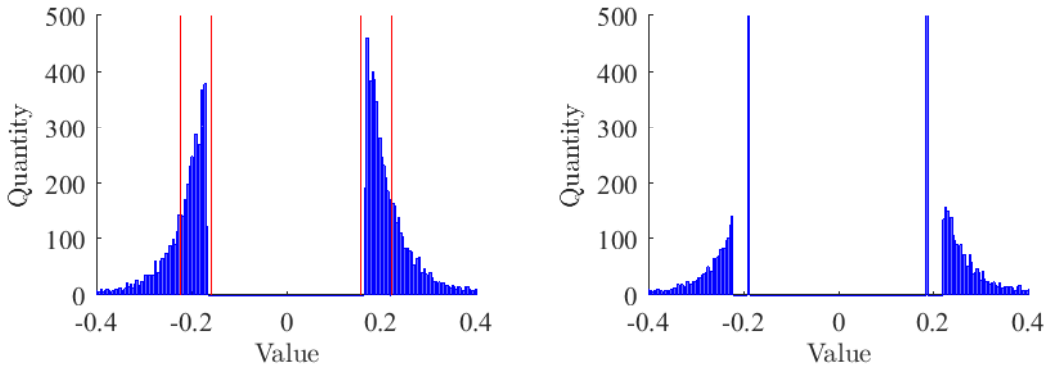


Figure 4.1: Starting point and goal of clustering. The 0 peak is omitted for readability.

In this way, all the edge-weights will maintain their values, while the concentrated ones will be unified to their representative center (Figure 4.1).

This approach can also be seen as an application of the abstract idea that stands behind the pruning technique: all the weights around a particular value are compressed towards it, as they are going to be approximated. In this case, two symmetrical centers will be considered.

However, there is another reason to cluster these values, which will be better explained in the compression-dedicated chapter (see Chapter 6). Basically, this simplification allows the normalization of all the weights so that the two clustering centers become 1 and -1; multiplying the input values by the inverse of this normalizing factor, the result of weights-input multiplication will remain the same. Therefore, the resulting network will be a compromise between a binarized - zero or (minus) one - network, with all its advantages, and a full precision one, with its edge-weights untouched.

It is important to state that this technique is applied after the pruning stage has been performed. In this way, the two peaks are visible: if not, this algorithm would simply perform a blind clustering, without any reason to support its choices. Therefore, once chosen the pruned model where to start from, the clustering procedure will be applied to further compress it, without touching the pruned weights.

4.1.1 Previous works

Previous works often stop altering the weights distribution after the pruning stage. Still, other ones modify it in a soft manner or offer different approaches to continue simplifying it: for example, with weight quantization [1]. With this technique, proposed in different modalities, weights are substituted by simpler, lower-precision values spread throughout the distribution, so that they can be represented on a lower number of bits; the original value is then approximated by some formulas.

The proposed technique does the opposite: most of the values are approximated by a single one or, to be precise, by two symmetrical ones. So, most of the values will have a single (double) sample, while the remaining, external ones will have multiple samples - themselves.

In [1]'s work, weights are balanced for parallel computation: some previously pruned weights are restored and other ones are set to 0 for the sake of a regular matrix structure; in [23]'s work, this regularity is accurately studied. In general, networks are distortion-resistant, as stated by [24], allowing the movement or migration of some weights for savings purposes: so, these remarks were indeed the foundations of this work.

4.2 Clustering formalization

The core of this procedure can be represented by the following sentence: if a weight has a value *close to* the *clustering center*, that value will be set equal to the clustering center itself. Of course, a formal definition for both the emphasized terms must be introduced.

4.2.1 Clustering centers

First of all, the two clustering center λ_1 and λ_2 will be symmetrical, thus meaning that $\lambda_1 = -\lambda_2 = \lambda$. This choice is due to the shape of the distribution and to the normalization reason mentioned above. Therefore, from now on, all the reasoning will be performed with positive weights, *i.e.* on the positive side of the distribution, and then mirrored.

In this technique, the clustering center λ is a parameter, thus its choice has to be performed carefully and will have a certain influence on the outcome; this weakness will be overcome by an evolution of this method shown in Chapter 5. Nevertheless, it is easy to see that the two peaks of the distribution are necessarily close to the pruning thresholds found in the previous pruning stage: it means that the clustering center can be set close to τ, if not to τ itself. To be more general as possible, the degree of freedom is transferred to a more meaningful parameter, which is the *threshold shift* δ. In this way, λ becomes:

$$\lambda = \tau + \delta$$

Usually δ will be small and positive: during the pruning phase, only few values will re-enter the pruned range below τ, meaning that the candidate center λ will be just to the right of τ. This is true either if we want to choose the mode of the distribution as λ or take a more weighted mean sample inside the peak-dominated range (Figure 4.2).

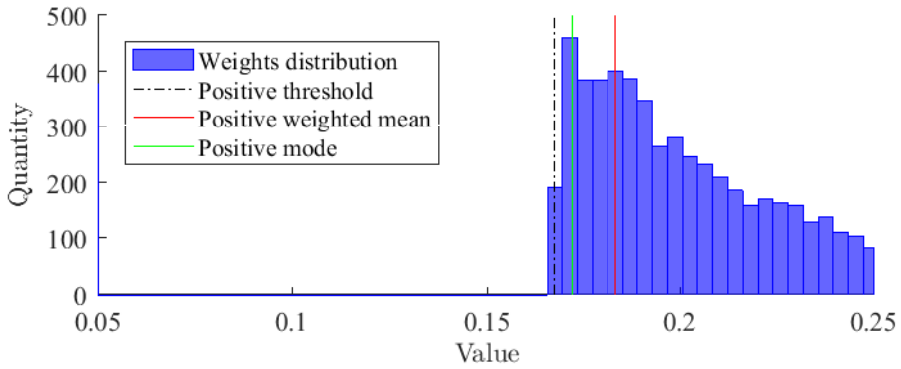


Figure 4.2: Choice of λ.

4.2.2 Clustering radius

Once δ is chosen, the procedure can start clustering weights that are close to λ. The definition of *close to* relies on the introduction of another parameter: the radius ρ. All the positive weights w_j belonging to the interval $[\lambda - \rho; \lambda + \rho]$ are close to λ (Figure 4.3).

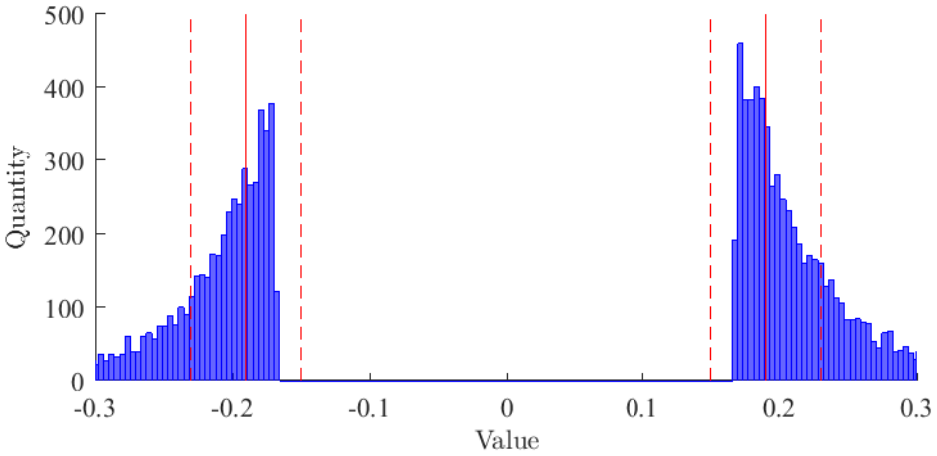


Figure 4.3: Clustering radius and ρ-closeness.

We can also state this sentence using the concept of distance: all the weights w_j with a distance from λ lower than or equal to ρ are close to λ. For we are speaking about one-dimension Euclidean distance, its formula can be written in the following way:

$$\sqrt{(w_j - \lambda)^2} = |w_j - \lambda|$$

So, a weight $w_j > 0$ is ρ-close to λ if this condition holds:

$$|w_j - \lambda| \leq \rho \tag{4.1}$$

This condition is valid only for positive weights because λ is equal to λ_1, which is the positive clustering center; instead, negative weights must get clustered not because of their ρ-closeness to λ_1, but because of their ρ-closeness to the negative one, $\lambda_2 = -\lambda_1 = -\lambda$. Thanks to this symmetry, saying that a negative weight w_k is ρ-close to $-\lambda$ is equivalent to say that its absolute value $|w_k|$ is ρ-close to λ.

Therefore, we can extend (4.1) also to negative weights, without having two separate formulas for the two clustering centers: a generic weight w_j is ρ-close to λ or $-\lambda$ if:

$$||w_j| - \lambda| \leq \rho \tag{4.2}$$

The choice of ρ is again to be performed carefully: but in this case, the algorithm can try several values and pick the one not worsening the network accuracy, making a compromise between it and the amount of clustered values - which means compression and operations savings, as explained in Chapter 6.

4.2.3 Clustering formula

The simpler sentence introduced at the beginning of this section (section 4.1) can now become a more complete and formal statement to explain the clustering procedure: all the (positive) weights which are ρ-close to λ will be clustered to λ; all the (negative) weights which are ρ-close to $-\lambda$ will be clustered to $-\lambda$; as previously said, pruned weights will remain pruned; finally, all the other weights will keep their value. This can be summarized up into the following, final clustering formula:

$$Cluster(w_j) = \begin{cases} 0, & \text{if } w_j = 0 \\ sign(w_j)\lambda, & \text{if } w_j \neq 0 \text{ and } ||w_j| - \lambda| \leq \rho \\ w_j, & \text{otherwise} \end{cases} \tag{4.3}$$

4.3 Clustering time and modality

As it is for pruning (see Chapter 3), this technique will be applied together with a short retraining to be sure that accuracy keeps stable. Again, a slowing factor γ_c is introduced into the update formula of the gradient descent algorithm, similarly to equation (3.3):

$$\vec{w}_{t+1} = \vec{w}_t - \gamma_c \vec{\eta} \odot \nabla \mathcal{L}_M(\vec{w}_t) \tag{4.4}$$

and its value can be around 0.05, as found empirically.

Furthermore, in order to be as soft as possible, the step-by-step approach will also be used here: the radius ρ will be substituted by a sequence of radiuses $\{\rho_k\}_{k \leq S}$ linearly increasing at each step k, and their limit ρ_S will be set to λ in order to bettere explore all the possibilities. So, each clustering step will start from the result of the previous one, building a progressive clusterization around the same λ.

For the same reasons shown for pruning, also the clustering centers can be considered separately for each matrix. However, as their value would be arbitrary anyway, this approach was not explored: instead, it evolved in a more sophisticated technique, precisely based on the importance of λ's position, which will be explained in Chapter 5.

Aside, another degree of freedom can be used, where two alternative are possible: clustering can be revocable or irrevocable. Finally, the optimizator can interact in three different ways with the clustering procedure.

All of these aspects will be accurately discussed below.

4.3.1 Revocability

4.3.1.1 Irrevocable clustering

This approach is based on the same idea of the pruning technique. In this case, the (4.3) formula is applied once at the beginning of each step: thus, the model gets clustered once. All the weights ρ_k-close to the two clustering centers are permanently set to $\pm\lambda$; then, the remaining non-pruned weights evolve as normal during the retraining phase to compensate this change. This is too much restrictive.

4.3.1.2 Revocable clustering

This approach can be seen as the most general one. Here, clustering is done *during* the retraining phase. It means that the clusterization is performed after each update, *i.e.* for each mini-batch, depending on the evolution of weights. In this way, the ρ_k-closeness is checked not simply for w_j, but for $w_j + \Delta w_j$: in other terms, the condition is evaluated after the (4.4) is applied by the optimizer, where for the sake of brevity

$$\Delta w_j = -\gamma_c \eta(w_j) \frac{\partial \mathcal{L}_M}{\partial w_j}(w_j)$$

With this formalism, considering w_j as the weight at time t, while $w_j + \Delta w_j$ is the weight at time $t+1$, the clustering formula (4.3) becomes

$$Cluster(w_j, \Delta w_j) = \begin{cases} 0, & \text{if } w_j = 0 \\ sign(w_j + \Delta w_j)\lambda, & \text{if } w_j \neq 0 \text{ and } ||w_j + \Delta w_j| - \lambda| \leq \rho \\ w_j + \Delta w_j, & \text{otherwise} \end{cases} \tag{4.5}$$

To be precise, if $w_j = -\Delta w_j$, this formula would also prune weight w_j; practically this case cannot happen, as weights do not move so much during retraining, also because of the slowing factor. Even if it was possible, it would mean that w_j arrived to zero because the loss was - at least locally - lower, and another pruned weight can just improve compression. However, it is still possible to keep track of pruned and not pruned weights to avoid this problem; of course, performing these checks requires memory and time.

With this formula, which can be considered as a complete update expression wrapping the gradient descent one, weights can exit or enter the two clustering intervals, so that they can better adapt to this new forced shape.

When ρ is reasonably small, *AdaDelta*'s gradients accumulation can eventually allow a weight to jump far than ρ itself, if necessary. Of course, the same is true for a weight on the edge of the interval pushed in: it will be immediately clustered into $\pm\lambda$.

4.3.2 Clustering awareness

The gradient descent algorithm can be informed or not about the clustering procedure. This distinction stands in the way in which the clustering formula interacts with *AdaDelta*'s updates: they can be mixed with different degrees of entanglement.

In particular, the *AdaDelta* optimizator can be

- unaware of the clustering procedure;
- totally aware of the clustering procedure;
- partially aware of the clustering procedure.

4.3.2.1 Unawareness

The *unaware* approach is quite equivalent to the irrevocable clustering. If the updates are not playing any role in the clusterization, it is useless to apply the clustering formula (4.5) for each mini-batch. Weights will evolve normally during the retraining phase; of course, clusterized weights must be kept to $\pm\lambda$ or all this phase would be useless. Again, this is not a good choice.

4.3.2.2 Total awareness

The *totally aware* approach is instead quite equivalent to the revocable clustering. Here, weights must follow all the restrictions imposed by the clustering stage; gradients will be computed on $\pm\lambda$ for weights not able to escape the attraction area with a single update, because they would return there. It is also possible to add a regularization term to the loss function to represent that a movement lower than ρ would have no immediate effect on a clusterized weight, while a little movement towards the clusterization intervals would imply a jump on $\pm\lambda$; this expedient would led to more meaningful and informed gradients, but would also be difficult to insert without adding discontinuous - and so not derivable - terms to the function. Instead of doing so, similar wrong situation will be eventually

corrected by subsequent runs of the update algorithm, which will extract or re-insert a weight into the clusterization areas: auto-correction can substitute this information.

4.3.2.3 Partial awareness

The *partially aware* approach is based on the idea of having two separate copies of weights: $\vec{w}_0$, which can assume all the continuous values, and $\vec{w}_\lambda$, which is restricted to the discretization operated by the clustering procedure in the two intervals around $\pm\lambda$. At the beginning, $\vec{w}_0$ and $\vec{w}_\lambda$ would be equal because clustering formula (4.3) would have been applied on weights to start retraining. Then, $\vec{w}_0$'s elements will start to evolve normally, without any kind of restriction, following the proper gradients indications.

Gradients will be computed in $\vec{w}_0$, but applied to update $\vec{w}_\lambda$ elements too: this is very different from the normal approach and the reason is the following one. Weights on $\vec{w}_\lambda$, which are the ultimate target of this phase, will move with the same gradients of $\vec{w}_0$, but they will also be clusterized, possibly obtaining a different result from the ones reached by unaware and totally aware clustering. In fact, in the totally aware clustering all the gradients are computed in and applied to the $\vec{w}_\lambda$ point; in the unaware approach, gradients are computed in and applied to $\vec{w}_0$, while $\vec{w}_\lambda$ is created as a clusterized copy of it. Here, instead, $\vec{w}_\lambda$ is updated as

$$Cluster(W_\lambda, \Delta W_0) \neq Cluster(W_\lambda, \Delta W_\lambda) \tag{4.6}$$

To imagine why this inequality holds, it is enough to think about the simple update:

$$W_\lambda + \Delta W_0 \neq W_\lambda + \Delta W_\lambda$$

This is true only after the first update: there, $W_0 = W_\lambda$ and so $\Delta W_0 = \Delta W_\lambda$, but thereafter W_λ is clusterized and so it starts a different journey ($\Delta W_0 \neq \Delta W_\lambda$) as derivatives are computed in different points.

This approach is quite complex and this partial blindness is risky. First experimental results were indeed not so good; eventually, for this approach was difficult to be justified, it was discarded: gradients would have been applied far from their neighborhood of validity, as explained in Chapter 2. Therefore, their updates are not guaranteed to be meaningful or right. A similar problem is further and better discussed in Chapter 5 for spiking.

4.4 Results

As said, revocable and total aware clustering was chosen for its best performances. The following discussion and analysis of results will refer to it.

4.4.1 Weights

The evolution of weights distribution during some of the steps is shown in Figures 4.4 and 4.5. Weights escape from or enter to the clustering region during the retraining; as the radius enlarges, most of the escaped ones will be re-clusterized during the following step. Zero peak is removed and the $\pm\lambda$ ones are cut to better show the remaining weights.

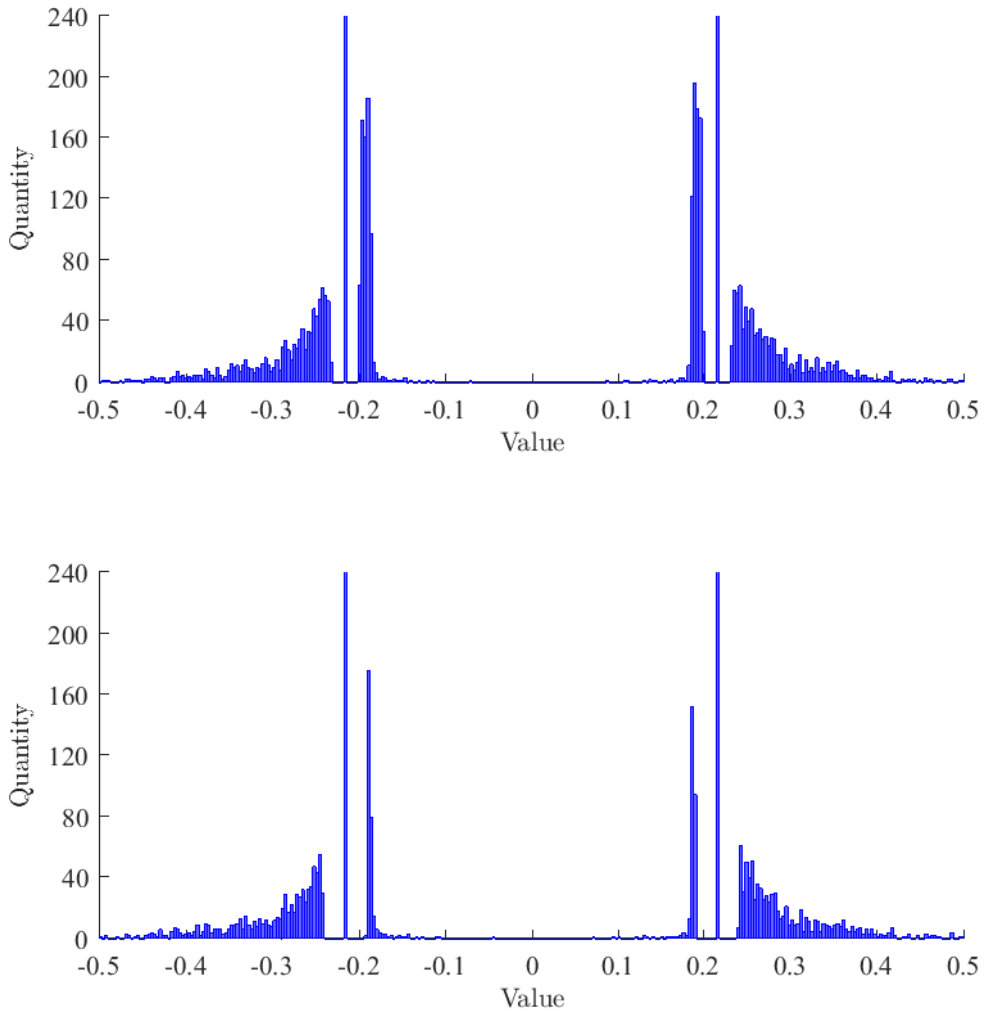


Figure 4.4: First steps of clustering.

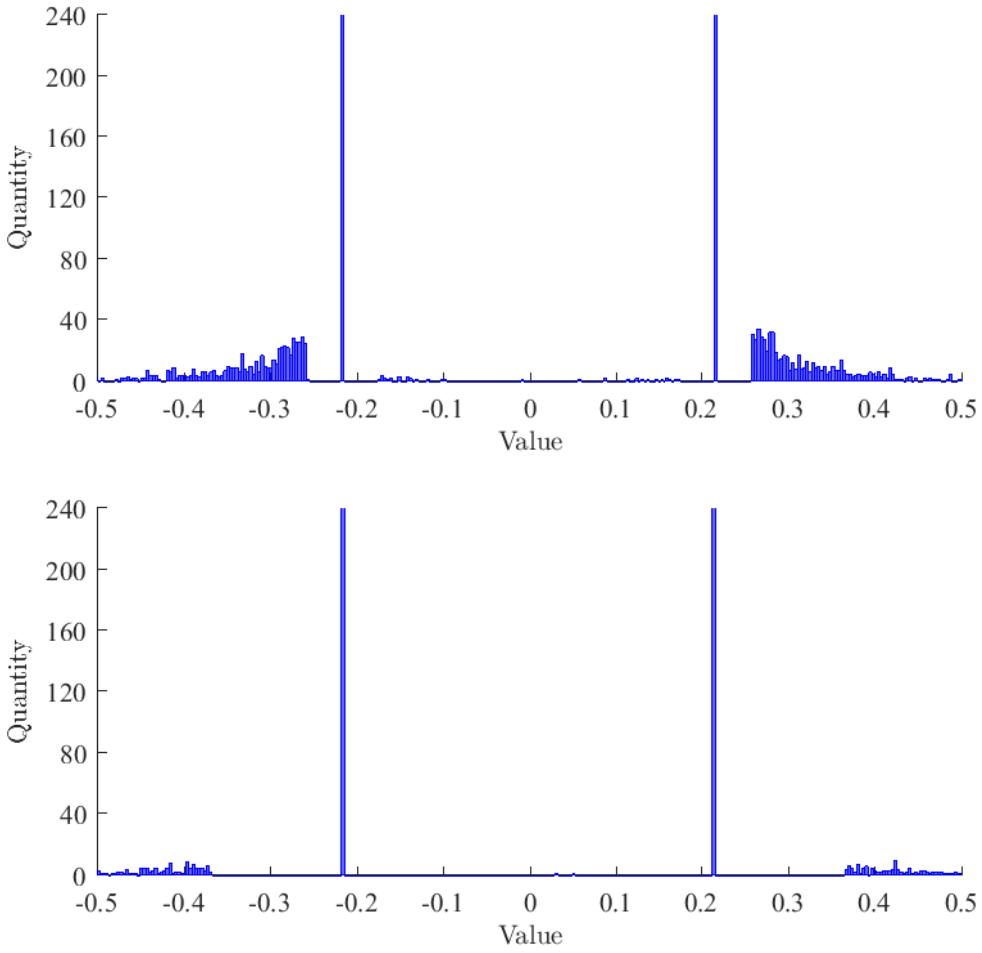


Figure 4.5: Last steps of clustering.

4.4.2 Accuracy

Even if it often improves over the original training accuracy of type A, clustering results show a little worsening with respect to the starting pruned model, which was taken with p almost equal to 95% - usually the optimal value for type A. Figure 4.6 shows this behavior, where δ was set to 0.03, but this value is quite arbitrary and depends on the closest local minimum.

Even if it was an improvement over the training accuracy, the clustering technique evolved in something with more solid mathematical bases, as explained in the following chapter. For this reason, this method was not tested on the other types; furthermore, its resulting compression would have been less powerful than the spiking one (see Chapter 5).

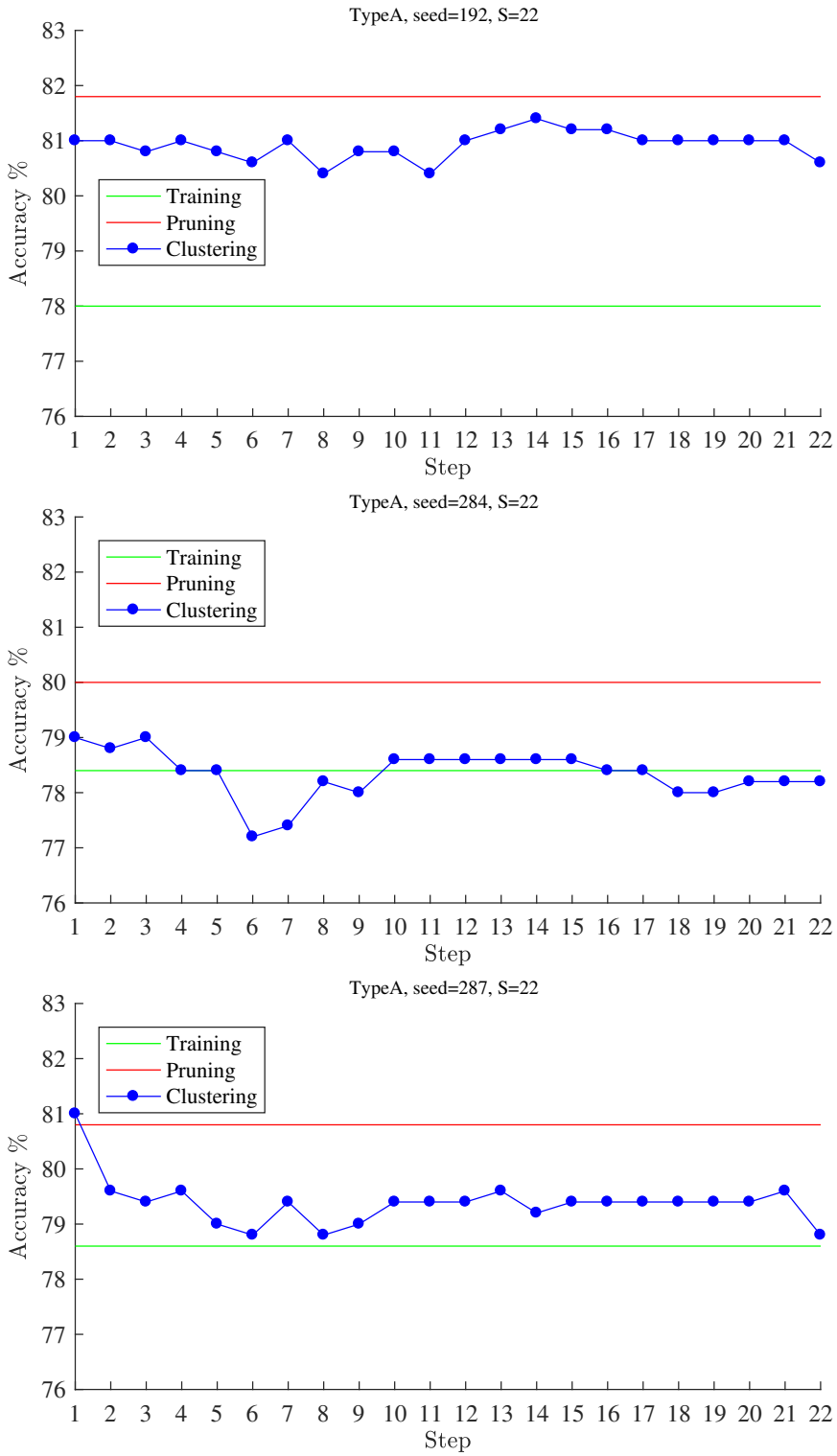


Figure 4.6: Clustering results.

Chapter 5

Spiking

5.1 Motivation

Even if not astounding, the results of the clustering method seemed quite good in an accuracy-compression compromise perspective: a local, acceptable minimum has always been reached, meaning that the weights collapsing approach was working. So, several ideas arose to improve this methodology.

Looking at the distribution of weights after the clustering stage, and in particular for high values of ρ, it is possible to notice the conglomerations of edge-weights standing beyond the clusterization areas. This result was not only predicted, but also wanted: it was done to respect the presumed importance of high-valued weights. But what would happen if this concept was abandoned?

As a first step, instead of totally drop them, it could be possible to cluster the remaining weighs in other two groups, using a secondary clustering procedure to collapse them around a couple of opposite centers. In this way, the distribution would show five peaks, meaning that just five values would be possible for weights (Figure 5.1).

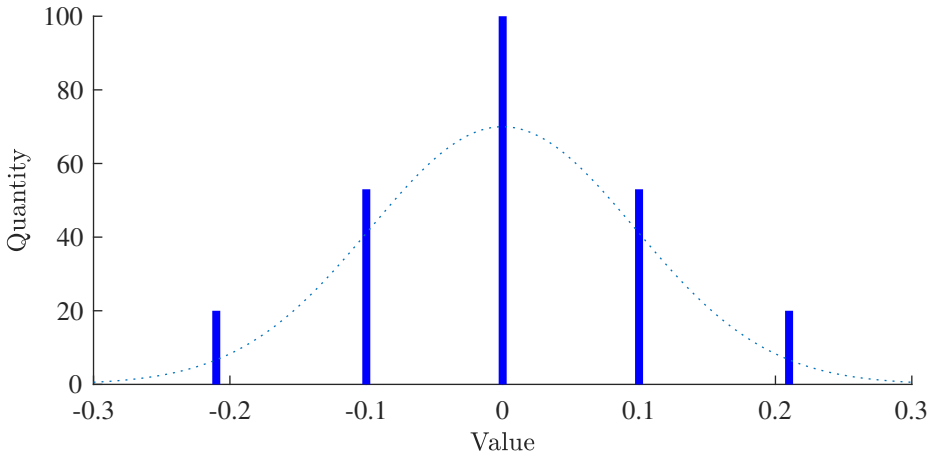


Figure 5.1: Five peaks of secondary clustering.

Of course that would led to a significant compression, if done with a proper encoding, which

was the guideline of this improvement effort and research. Still, stronger approaches and ideas are possible, which do not require the introduction of new parameters such as the secondary clustering centers. For example, it could be possible to enlarge on one side the clusterization areas in order to include the edge weights; however, in this case, the clustering formulation with its symmetry around λ and the peaks would not be adequate anymore. Therefore, the same problem arises: how to choose the new clustering center.

Such observation was the primary reason that led to this new technique: a smarter method that can *find* which are the optimal centers where to collapse all the non-pruned weights, in order to obtain a distribution with just two pronounced *spikes* aside from the pruned elements.

This method is a substitution and an evolution of the clustering technique, which means that it will start from the pruned model to find the best *spiking centers* to use.

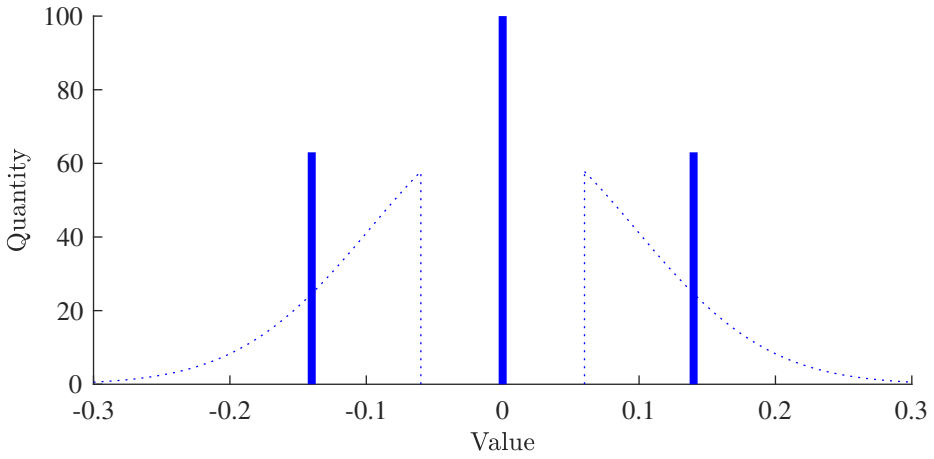


Figure 5.2: Finding the spiking centers: three peaks with the 0s one.

The secondary aspect of this method is the simplification of weights into their two primitive functions, which are the inhibitory and excitatory ones. As Figure 5.2 shows, three values will be possible for weights: zero, σ and $-\sigma$, calling $\pm\sigma$ the spiking centers. Furthermore, if weights are normalized, it is possible to obtain a great result: a *ternarization* of the model, where weights can only be 0, 1 or -1, with all the benefits implied. All these aspects will be better explained in the dedicated chapter Chapter 6.

5.1.1 Previous works

This technique, as evolution of the clustering one, finds its inspiration in the same previous works. Also, it is very similar to a binarization, like in [25]'s work, where weights are fully

binarized to $+1$ and -1: while this approach was initially believed impossible, or not able to keep a stable accuracy, recent works have demonstrated the opposite.

In particular, after this technique was fully developed, a similar algorithm has been found in literature [7], applied for a different network - a convolutional one - and with different purposes and motivations: as explained below, in this work the spiking center is updated thanks to a weighted sum, which is a scalar product, instead of a simple average like in [7], even if the mean itself was the conceptual starting point; furthermore, there different values are found for each layer and filter, making the general extraction of a single spiking center impossible - and all the advantages it implies. A fundamental difference stands also in the way the weights are updated. Here, gradients and updates are computed and applied always to the same spiked matrix, because weights are immediately set to σ, without the hard-to-justify mix of having a normal copy of matrices running thanks to the gradients computed on other points, as it was for the discarded approach for clustering: indeed, moving a weight on the direction found into another point can possibly take to a maximum, or in general will not lead surely to a minimum. In this technique, instead, the directions are computed regularly and are projected on the restrained searching space.

Also, this spiking technique is applied after a training stage and a pruning phase to learn which weights are significant, and it requires a little number of epochs, because it is aimed also to improve network performances, if possible. Furthermore, it can be applied independently from the methodology used to initially train the network and, in future works, this approach will be modified to find the better weighting strategy for the updates, as explained below. Instead, in [7]'s work, the network is directly trained with these mixed gradients.

5.2 Starting point

To learn the position of the spiking center σ, the gradient descent algorithm will be used again, and in particular its *AdaDelta* version. However, differently from the previous methods, its ability to find a local minimum will not be used to retrain and repair the model, but as a searching guide to move into a constrained solution space. In other words, this method will directly find σ while minimizing the loss function, using the information provided by *AdaDelta*, instead of a blind and exhaustive search throughout all the possible values. For this reason it could also be seen as a new constrained version of the SGD algorithm and *AdaDelta* itself, in some ways. In this sense, it is a great improvement over the clustering technique, which has an arbitrary choice of its centers.

Also, the step-by-step approach has no meaning here. Thus, only few epochs will be required for the algorithm to find σ. In the terms of previous techniques, just a single retraining will be necessary.

Again, a slowing factor γ_s will be introduced, better discussed in section 5.4.

5.2.1 Primordial spiking procedure

The spiking center is found by using all the information available: initially, this was done by a simple *arithmetic mean*, which instead hides several fundamental *weighted* aspects explaining its effectiveness: they will be introduced with the complete new formalization. Here, the primitive version will be shown.

First of all, because of the intrinsic symmetry of the problem, the absolute values of updated weights $|w_j + \Delta w_j|$ is taken. Signs are discarded because, of course, computing an arithmetic mean on quite symmetrical and opposite values would result in a spiking center close to zero, which is not useful. Also, the spiking center σ must represent not only the positive or negative half of weights, but all of them, and so the absolute value allows them to equally participate in a single computation instead of having two possibly different means for the positive σ_1 and negative σ_2: the identity $\sigma = \sigma_1 = -\sigma_2$ must hold, as it was for λ, otherwise the idea of having equal inhibitory and excitatory strengths would not be satisfied and all normalization process could not be applied anymore. By performing a single global computation, this equality is automatically reached.

After that, the mean can be applied to obtain the positive spiking center. It is important to explicitly state that pruned weights are excluded from this computation: otherwise, their contribution would be a huge bias. It would shift back σ, making the non-pruned weights unable to significantly participate into the mean. This is especially true for highly pruned models, where the $90 - 95\%$ of weights are zero.

Then, exactly as it is for clustering, all the weights are set to σ or $-\sigma$ according to the signs they had, and the spiking process continues for each update, *i.e.* for each mini-batch.

To summarize,

1. Take the non-pruned weights;
2. Compute their absolute value;
3. Compute their arithmetic mean;
4. Set them to $\pm\sigma$ according to their signs.

Figure 5.3 shows this process in a graphical way.

As explained below, this process will be substituted by a smarter one (see section 5.3), which is its natural evolution.

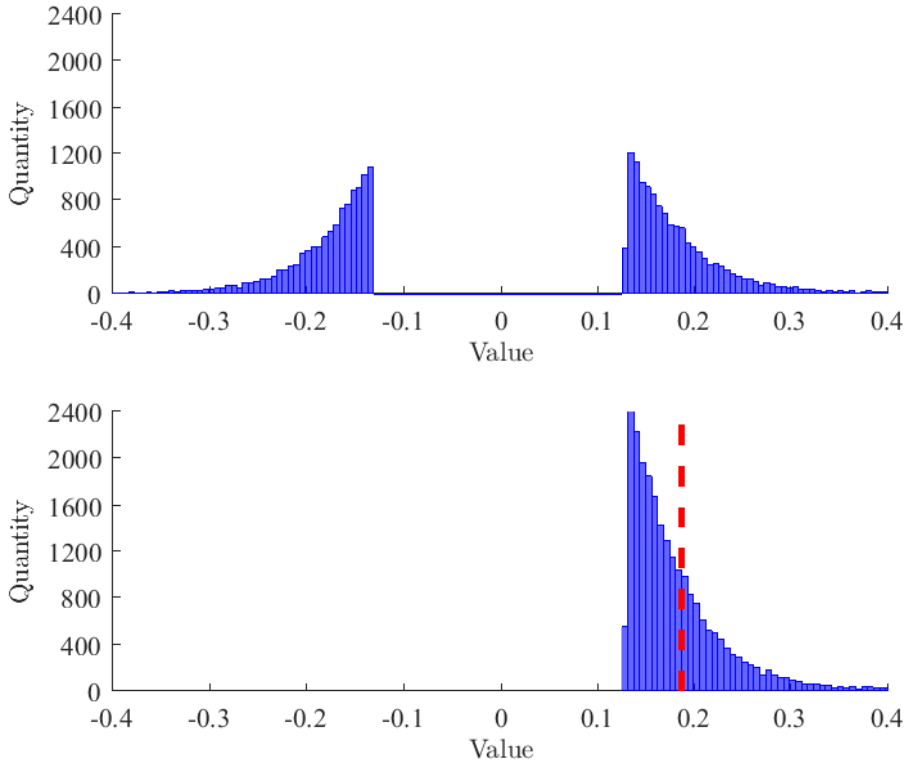


Figure 5.3: Primitive procedure to find σ.

5.2.1.1 Grouping matrices

When networks have more than one matrix, like LSTMs and GRUs, a single σ is required for each of them to obtain matrices with only 0, 1 and -1 with the normalization process. Each matrix will have weights $\in \{-\sigma;\ 0;\ \sigma\}$, possibly with different values of σ.

However, this restriction is not mandatory. It is possible to perform the spiking average on groups of matrices to find a common σ. The number of spiking centers σ is strictly related to the number of operations savings, so this idea can lower them.

To be precise, operations are saved only if the n matrices to be grouped multiply the same vector. See Chapter 6 for further details.

Experimental results show that this approach is valid (see section 5.4). For example, it is possible to group matrices with similar jobs, as they have similar distributions. In the LSTM case, grouping the four input matrices W and doing the same with U matrices

is a good idea; instead of having eight different σ values, just two will be used. As similar distributions and jobs will reasonably lead to very similar σ values, it is useless to keep four distinct spiking centers which differ by a tiny number.

To be precise, even this two σ values were very similar, but the W matrices and the U matrices multiply two different vectors, $\vec{x}$ and $\vec{h}$, meaning that a further grouping would not have saved any other operations.

5.2.2 Simple spiking formula

Formally, the initially found spiking procedure can be written as:

$$Spike(w_j, \Delta w_j) = \begin{cases} 0, & \text{if } w_j = 0 \\ sign(w_j + \Delta w_j)\sigma, & \text{if } w_j \neq 0 \end{cases} \tag{5.1}$$

It is very simple because all the complexity is hidden by σ, which conceals the details of its computation as:

$$\sigma = \frac{1}{N_s} \sum_{j=0}^{N_s-1} |w_j + \Delta w_j| \tag{5.2}$$

This is the arithmetic mean introduced above, where $N_s < N_t$ is the number of non-pruned weights $\{w_j\}$. Remembering the definition given for Δw_j (**??**), equation (5.2) above becomes:

$$\sigma = \frac{1}{N_s} \sum_{j=0}^{N_s-1} \left| w_j - \gamma_s \eta(w_j) \frac{\partial \mathcal{L}_M}{\partial w_j}(w_j) \right| \tag{5.3}$$

η's properties and the derivative-dependent updates are part of the motivations why this spiking procedure works so well (see section 5.4 for results) as explained in the dedicated section below.

5.3 Spiking formalization and effectiveness

Even if the initial choice of using the arithmetic mean can seem arbitrary or too much simple, it is not. The true spiking formula, which is a scalar product, will explain why.

This method inherits all the good properties held by *AdaDelta*, and SGD algorithm in general. It is a training in full, with an appropriate and particular initialization, if we consider the previous phases as a black box initializer.

Three are the main reasons:

- The hidden weighted formula behind the arithmetic mean;

- The (huge) amount of pruned weights;
- The stochasticity of mini-batches.

While stochasticity and its ability to escape from only-locally optimal solutions are the same considered for general SGD, thus already explained in Chapter 2, the other two reasons are particular for this method and require a longer explanation.

5.3.1 True weighted spiking formula

The first reason is represented by the adaptive learning rate η and the gradient: each update is *weighted* depending on η and the gradient. Thus, each w_j moves in a *weighted*, different way. This inevitably results in a sort of *weighted* arithmetic sum of w_j, where one w_j can have more or less strength in pulling the mean σ towards itself.

To better formalize this idea, equation (5.2) can be written separating positive and negatives $w_j^* = w_j + \Delta w_j$:

$$\sigma = \frac{1}{N_s}\Big(\sum_{j:w_j^*>0}(w_j + \Delta w_j) \; - \sum_{j:w_j^*<0}(w_j + \Delta w_j)\Big)$$

which can be further manipulated:

$$\sigma = \frac{1}{N_s}\Big(\sum_{w_j^*>0} w_j \; + \sum_{w_j^*>0} \Delta w_j \; - \sum_{w_j^*<0} w_j \; - \sum_{w_j^*<0} \Delta w_j\Big) =$$

$$= \frac{1}{N_s}\Big(\sum_{j=0}^{N_s-1} |w_j| \; + \sum_{w_j^*>0} \Delta w_j \; - \sum_{w_j^*<0} \Delta w_j\Big) =$$

$$= \frac{1}{N_s}\sum_{j=0}^{N_s-1} |w_j| \; + \; \frac{1}{N_s}\Big(\sum_{w_j^*>0} \Delta w_j \; - \sum_{w_j^*<0} \Delta w_j\Big) \tag{5.4}$$

The first term of the last member (5.4) is the arithmetic mean of the previous weights' values w_j, *i.e.* before the Δw_j is applied. Aside from the first update, all the weights are always set to $\pm\sigma$: it means that the first term is equal to the previous σ, here called σ_t, because $|w_j| = \sigma_t, \forall j < N_s$.

$$\frac{1}{N_s}\sum_{j=0}^{N_s-1} |w_j| = \frac{1}{N_s}\sum_{j=0}^{N_s-1} \sigma_t = \frac{1}{N_s} N_s \sigma_t = \sigma_t$$

The second term of (5.4) can be seen as $\Delta\sigma$, the update of the spiking center. Therefore, this technique is finding the best σ applying the same principles used for w_j:

$$\Delta\sigma = \frac{1}{N_s}\Big(\sum_{w_j^*>0} \Delta w_j \; - \sum_{w_j^*<0} \Delta w_j\Big) \tag{5.5}$$

$$\sigma_{t+1} = \sigma_t + \Delta\sigma$$

which is equal to the update formulation used for w_j. Also, joining the two sums of (5.5) with a general term Λ_j instead of Δw_j to keep into account the right sign, and extracting $\eta(w_j)$ to underline it $\Lambda_j = \eta(w_j)\Lambda_j^*$, we obtain:

$$\Delta\sigma = \frac{1}{N_s}\sum_{j=0}^{N_s-1} \Lambda_j = \frac{1}{N_s}\sum_{j=0}^{N_s-1} \eta(w_j)\Lambda_j^*$$

So, the spiking method updates the spiking center with a weighted sum of Λ_j^*,

$$\sigma_{t+1} = \sigma_t + \frac{1}{N_s}\sum_{j=0}^{N_s-1} \eta(w_j)\Lambda_j^* \tag{5.6}$$

where, if $w_j + \Delta w_j > 0$ and $w_j = +\sigma_t$, with the definition (**??**) of Δw_j, Λ_j^* can be written as:

$$\Lambda_j^* = -\gamma_s \frac{\partial \mathcal{L}_M}{\partial w_j}(\sigma_t)$$

so it depends on the partial derivative of the loss function. The derivative is computed in $w_j = \sigma_t$ because if $w_j + \Delta w_j > 0$ it is unlikely that $w_j < 0$ ($w_j = -\sigma$): due to the slowing factor and because weights inhibitory or excitatory nature has already been learnt previously, weights practically keep their sign after the update. However, to be more general as possible,

$$\Lambda_j^* = -sign(w_j + \Delta w_j)\gamma_s \frac{\partial \mathcal{L}_M}{\partial w_j}(sign(w_j)\sigma_t)$$

but practically:

$$\Lambda_j^* = -sign(w_j)\gamma_s \frac{\partial \mathcal{L}_M}{\partial w_j}(sign(w_j)\sigma_t) \tag{5.7}$$

In this sense, (5.2) is not a simple arithmetic mean. As shown by equation (5.6) and knowing the formulation of η, σ is updated by a weighted arithmetic mean of gradients' components. So, if a weight is very far from its optimal value, its associated partial

derivative would be high, thus it will contribute in a sensible way to the mean and to σ's shifting.

In other words, averaging the weighted gradients will still be an optimum search, but instead of moving in a totally free space, weights will move on a constrained one, departing from or approaching to the origin - here the meaning of the absolute value. Indeed, Λ_j^* values will be positive if the actual point must get away from the origin, or negative otherwise.

However, properly, the constrained space is half of a straight line, thus a segment, and it is a sort of *semi-bisector*: its normalized direction is given by a vector $\vec{e}$ in the solution space defined as

$$\vec{e} = \frac{\vec{s}}{||s||}$$

where $\vec{s}$ is a vector whose components s_j are

$$s_j = sign(w_j)$$

The norm of $\vec{s}$ is:

$$||s|| = \sqrt{\sum_{j=0}^{N_s-1} sign(w_j)^2} = \sqrt{\sum_{j=0}^{N_s-1} 1} = \sqrt{N_s}$$

where N_s is the number of dimensions of $\vec{s}$, so the amount of non-pruned weights.

In practice, by writing W in its vectorized form $\vec{w}$ by columns or rows, as done for the training phase, all the weights, and thus the solution, can be found just along this direction, and it is

$$\vec{w} = \sigma_t \vec{s}$$

As explained above, some components of the direction can theoretically change, because w_j can potentially change sign: indeed, the solution space is the set of all the possible *semi-bisectors*, and the real definition of s_j is $sign(w_j + \Delta w_j)$, but for simplicity the explanation will concentrate on one of them only.

Figure 5.4 shows this remark in a graphical way. Supposing the original matrix is a simple 5×5 pruned until 2 positive weights w_x, w_y and 1 negative weight w_z are left (so with a reasonable 88% pruning percentage), the solution space can be represented in three dimensions and becomes a segment, whose direction is $\vec{s} = (1, 1, -1)^T$. If the actual solution $\vec{w} = (w_x, w_y, w_z) = (\sigma_t, \sigma_t, -\sigma_t)$ is in the attractive valley of the local minimum H, then $-\vec{\nabla}\mathcal{L}(\vec{w})$ will point it. However, $\vec{w}$ can only move in $\vec{s}$ direction, which is $\vec{e}$: so, it is possible to find the projection of $-\vec{\nabla}\mathcal{L}(\vec{w})$ on $\vec{e}$ to know which is the best movement $\Delta\sigma$ to get as close as possible to H: getting away from the origin or not. This is the ultimate

sense of this algorithm: lowering or increasing σ to get close to the minimum in the best approximated way. This projection can be represented in two dimensions anyway, because it lays on a plane with the semi-bisector: the solution space dimensions do not matter.

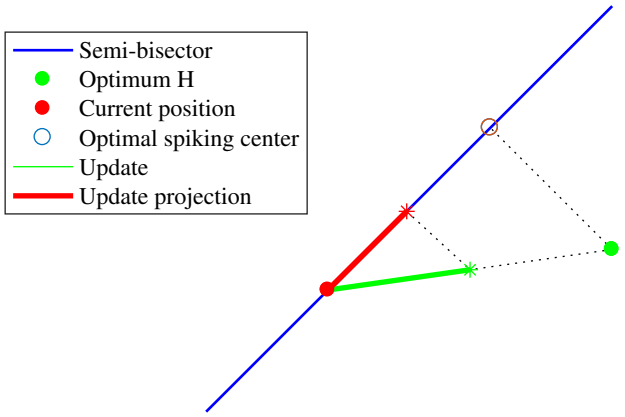


Figure 5.4: Projection on the semi-bisector.

The projection is found by using the *scalar product* between $-\vec{\nabla}\mathcal{L}(\vec{w})$ and $\vec{e}$:

$$\langle \vec{e}, -\vec{\nabla}\mathcal{L}(\vec{w})\rangle = \frac{1}{||s||}\langle \vec{s}, -\vec{\nabla}\mathcal{L}(\vec{w})\rangle = \frac{1}{\sqrt{N_s}}\sum_{j=0}^{N_s-1}(-s_j\frac{\partial\mathcal{L}}{\partial w_j}(sign(w_j)\sigma_t))$$

So, remembering the definition of s_j, it becomes:

$$\frac{1}{\sqrt{N_s}}\sum_{j=0}^{N_s-1}(-sign(w_j)\frac{\partial\mathcal{L}}{\partial w_j}(sign(w_j)\sigma_t))$$

Finally, if we add the mini-batch influence (the sum over M examples, by simply writing $\mathcal{L}_M$), the slowing factor for the gradient (γ_s), the learning rate parameter (η) because for the minimum search we are not simply using $-\vec{\nabla}\mathcal{L}_M(\vec{w})$ as in the simplified example, we exactly obtain the definition of $\Delta\sigma$ above, a part from a constant normalization by $\frac{1}{\sqrt{N_s}}$, once know the number of non-pruned parameters, which can be absorbed into the γ_s definition and tuning. In particular, the single element inside the previous sum becomes Λ_j^*, by confronting it with equation (5.7). Thus, $\Delta\sigma$ is now:

$$\Delta\sigma = \frac{1}{\sqrt{N_s}}\sum_{j=0}^{N_s-1}\left(\eta(w_j)(-sign(w_j)\gamma_s\frac{\partial\mathcal{L}_M}{\partial w_j}(sign(w_j)\sigma_t))\right)$$

So, the complete scalar product in vector and readable notation becomes:

$$\Delta\sigma = \left\langle \vec{e}, -\vec{\eta} \odot \gamma_s \vec{\nabla} \mathcal{L}_M \right\rangle$$

And the final, updating spiking formula becomes:

$$\sigma_{t+1} = \sigma_t - \left\langle \vec{e}, \vec{\eta} \odot \gamma_s \vec{\nabla} \mathcal{L}_M \right\rangle$$

Still, σ_0 will be computed with the first primitive formula, because of course no previous σ exists to be updated.

So, the optimality (or a saddle point) H' is reached not only when $\vec{\nabla}\mathcal{L}_M = \vec{0}$, which is the most lucky condition, but when

$$\left\langle \vec{e}, \vec{\eta} \odot \gamma_s \vec{\nabla} \mathcal{L}_M \right\rangle = 0$$

In other words, this happens when the point cannot move anymore along the semi-bisector because the only movement possible to reach the true H would be orthogonal to the permitted direction.

5.3.2 Amount of pruned weights

Starting from a pruned model is effective for two reasons.

First of all, all the useless weights have been cut. As explained in Chapter 3, they could have been also dangerous because of their possible ability to keep the model in a bad area of the solution space. Here they would be possibly dangerous too: because of their huge quantity, which is at least 90% of the total, they would heavily shift σ towards bad values, overcoming the influence of relevant weights.

Symmetrically, the small amount of non-pruned weights allows each of them to have a significant influence to the computation of σ, because the mean is calculated on a little set of values or, by a scalar product point of view, the dimensionality of the problem is reduced, so that each component can have an important effect on deciding to go far from or close to the origin.

As a proof of these observations, it is enough to see that this method produces the best result when applied to highly pruned networks (see section 5.4), which is a great property: it allows greater compressions and operations savings. In this sense, this technique can perfectly work together with the pruning method.

5.4 Results

This method leads to a perfectly spiked distribution, as desired. All the three peaks are shown in Figure 5.5.

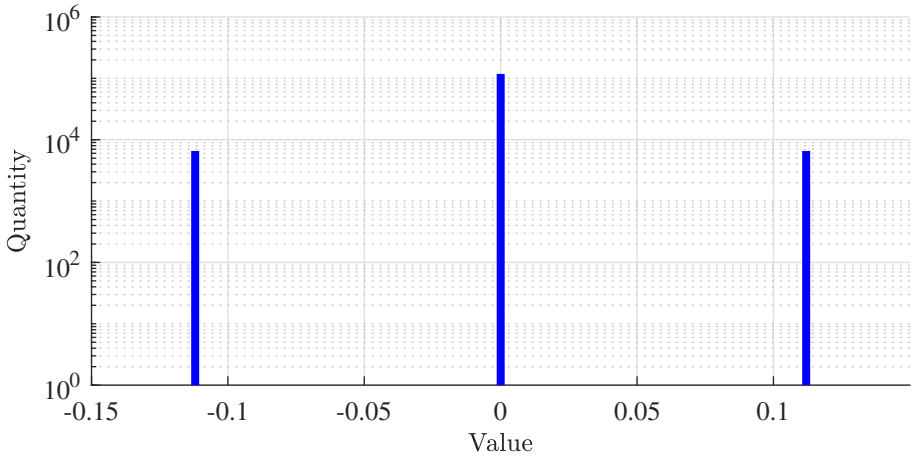


Figure 5.5: Weights distribution after spiking, 90% of pruning. Y axis scale is logarithmic because the two σ peaks are 20 times shorter than the 0 one: each one is an half of the 10% non-pruned weights.

Of course, σ value has some little variations from seed to seed, but the overall distribution is the same . The strength of this method stands in its absence of arbitrary parameters, apart from the slow factor γ_s, but its value is good when very low. For example, Figure 5.6 shows the γ_s-accuracy plot of a pruned model spiked with different γ_s.

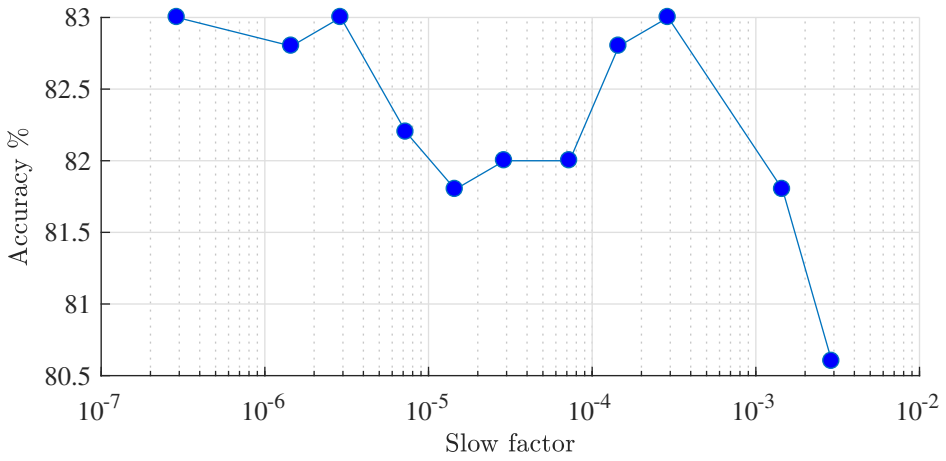
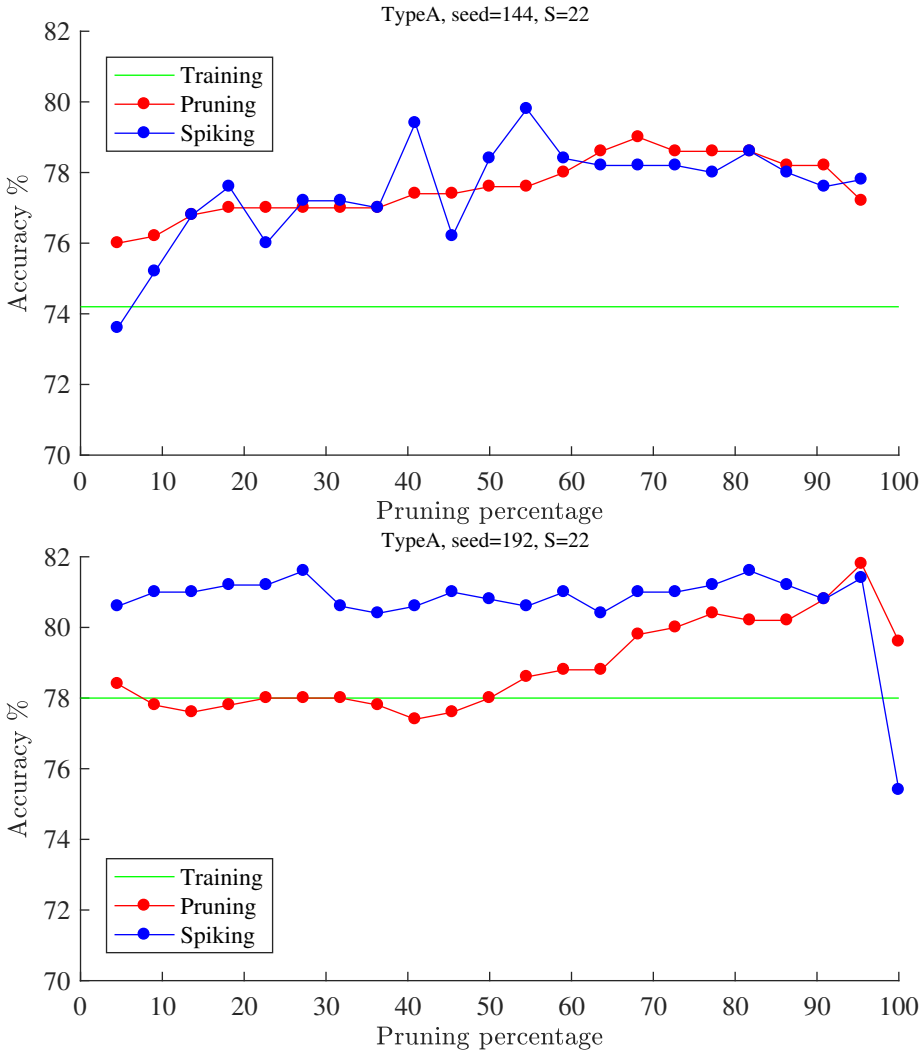


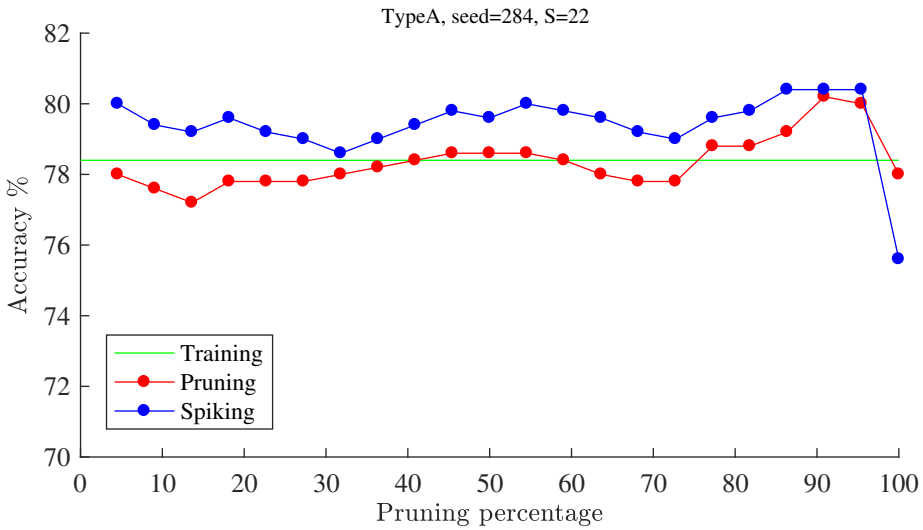
Figure 5.6: γ_s-accuracy plot.

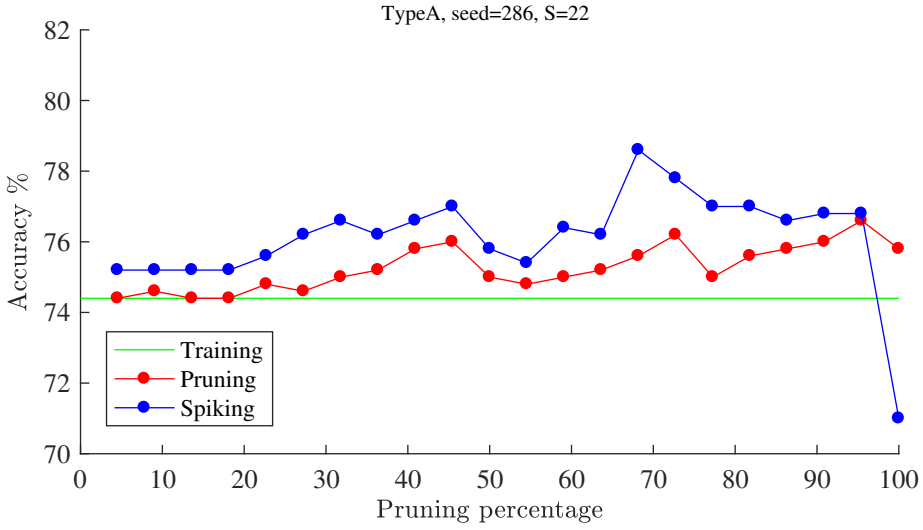
5.4.1 Accuracy

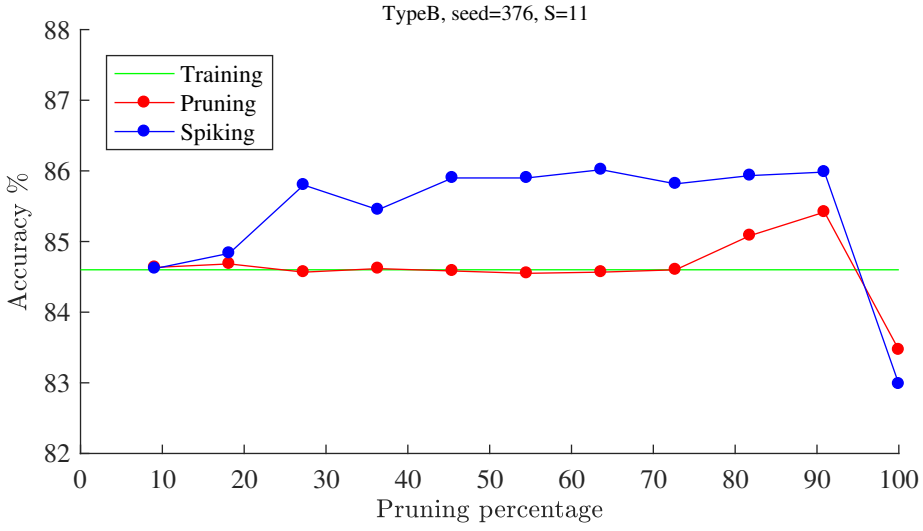
Accuracy keeps stable with respect to the pruning technique. In particular, before the 95% of pruning percentage, spiking method often improves the accuracy of the model. For type D, just the 90% pruned models have been reported, because spiking's improvement over compression are significant when the pruning percentage is high.

Spiking performs very well: for example, until type C (Figure 5.7), the accuracy is generally higher than the one of the normally trained model and pruned ones, even with the 90% of weights pruned away. Furthermore, for 99.9% pruned models, there are no significant degradations when the network is trained with the larger datasets B and C. When model compression is more important than 1% or 2% of accuracy loss, spiking method can achieve very good results.




TypeA, seed=284, S=22
Accuracy %
Pruning percentage
Training
Pruning
Spiking


TypeA, seed=286, S=22
Accuracy %
Pruning percentage
Training
Pruning
Spiking


TypeB, seed=376, S=11
Accuracy %
Pruning percentage
Training
Pruning
Spiking

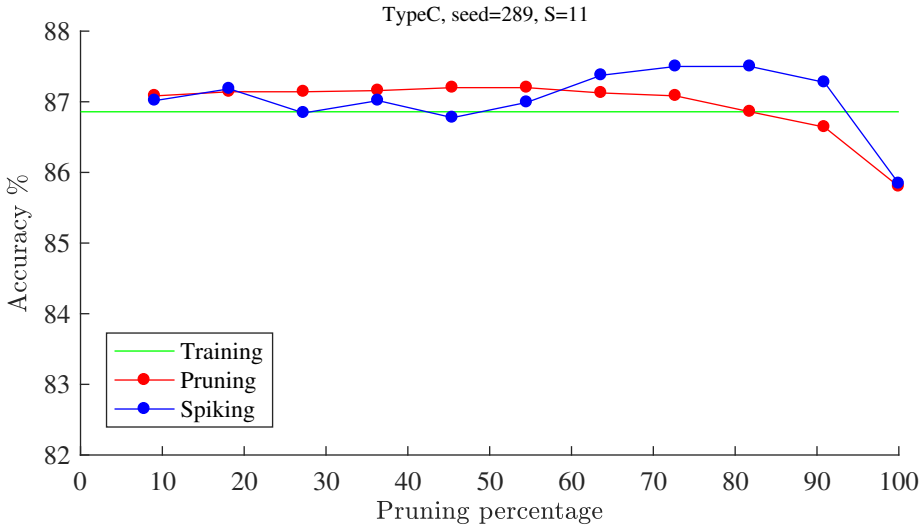


Figure 5.7: Spiking accuracies until type C (also in the previous pages).

For type D and pruning percentage 90%, accuracy keeps stable around 88.6% for various seeds, while the best model found had an only-training accuracy of 88.82% (seed 837). This technique can sometimes find a better minimum with respect to the previous phases, such in the case shown in Figure 5.8, but in general for $p = 90\%$ it shows a $0.2\% - 0.3\%$ loss with respect to only-training accuracy, which is still far more than acceptable.

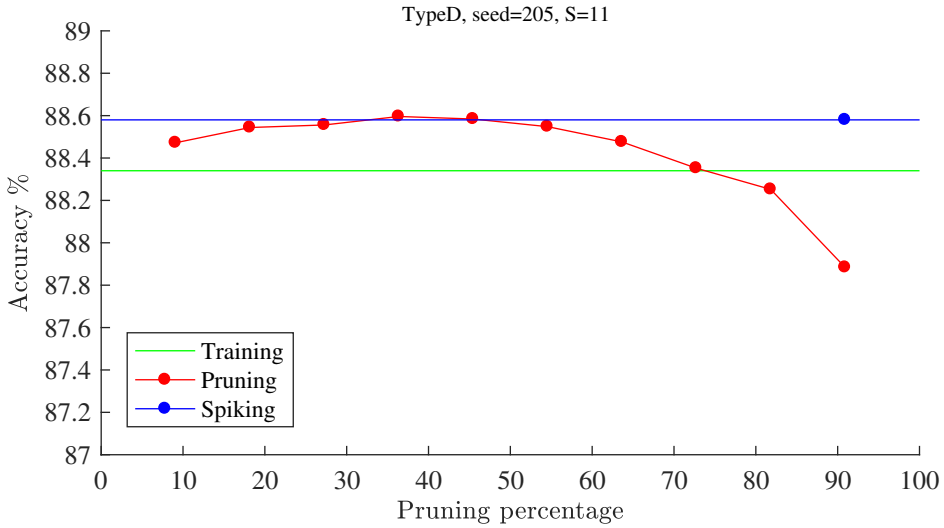


Figure 5.8: Particular spiking case: only the spiked 90% pruned model is shown.

Also another important result has been found: for type D, with a pruning percentage

of 81.8%, an accuracy equal to 88.9% has been found (Figure 5.9). This is state-of-the-art result, and slightly higher, as the best accuracy found in literature with this architeture is 88.89% [16]; also, as an intermediate result during the spiking phase, an 89.048% accuracy has been found. Probably, properly setting all the hyperparameters and with many runs (see Chapter 7), an even better result could have been reached.

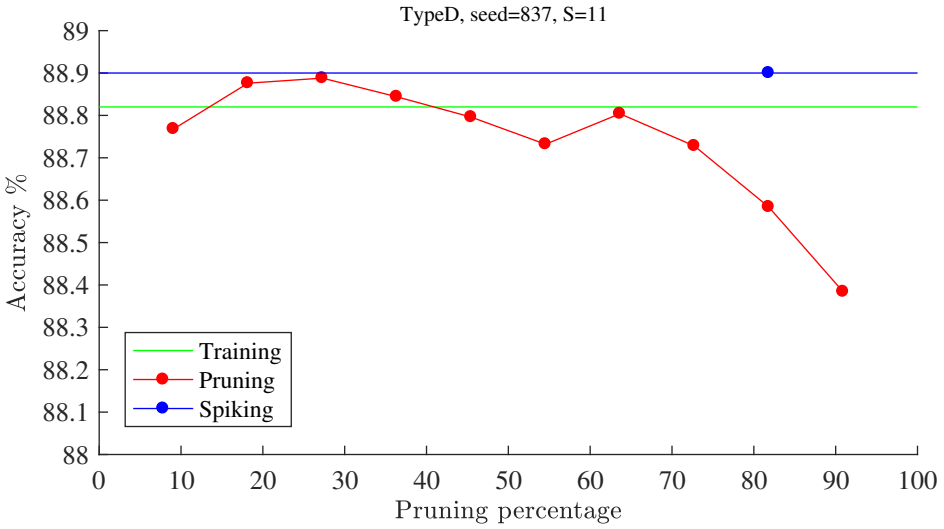


Figure 5.9: State of the art: only the spiked 81.8% pruned model is shown. The y scale is halved with respect to Figure 5.8.

To conclude, with $p = 99.78\%$, spiking has an accuracy slightly higher than simple pruned method (84.52% against 84.27%), as Figure 5.10 shows.

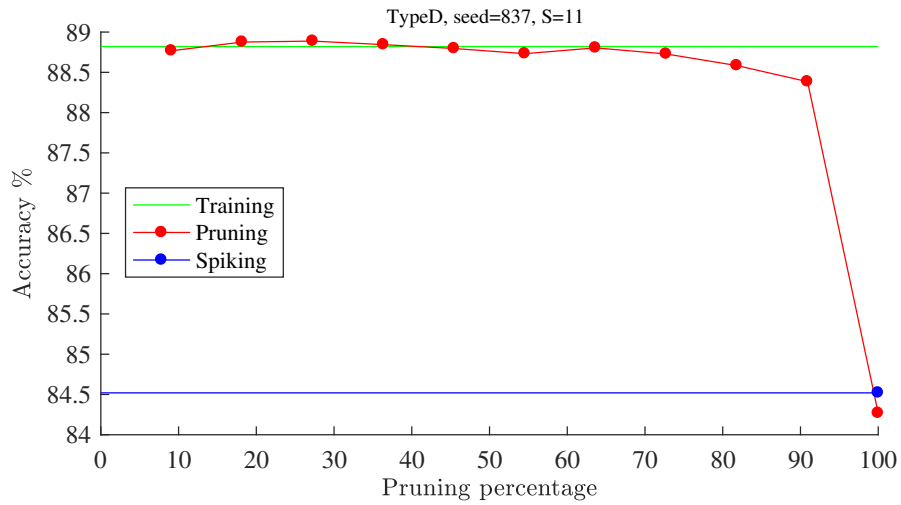


Figure 5.10: Spiking with 99.78% pruning.

Chapter 6

Hardware-oriented optimizations and compressions

6.1 Introduction

All the previous methods find their motivation and ultimate goal here. Their effort is aimed to reduce the number of a network's parameters or, better, to give the possibility of a smart implementation on a dedicated device.

The word *encoding* is used in this work to indicate the reduction of memory space and accesses required to store and get all the information about the network model, through the usage of clever codes and similarity patterns.

The word *savings* here refers to the elimination of a great quantity of operations or to an alteration of their order to obtain a smarter schedule for network's functionalities, which is useful to save computations.

This final stage of the overall process is a manipulation of the representation of networks. It can be done off-line because it does not require further retraining: it does not alter the model, but just its implementation and parameters storing.

It is important to state that even if this phase follows the other ones in practice, because it must be applied inevitably afterwards, it was the true starting point and the real motivation of the previous methods. They cannot be separated: spiking was developed because the compression shown in this work would have been very effective. Clustering and spiking methods themselves *are also* part of the compression in its most general sense. The encoding procedures and operations manipulations introduced below will exploit the advantages produced by them and will finally explain the reasons behind some choices.

All of these encodings are indeed thought to be easily and efficiently decoded by a hardware device, in order to reduce the overall number of memory accesses required to obtain the weights matrices, stored in little sized memories. Also, operations optimizations are meant to significantly reduce the amount of power consumption and time required by the dedicated device, using a smarter and quicker scheduling.

6.2 Previous encodings

The following one is a simple encoding already introduced by other works, such as [1], which can be used right after the pruning phase. This method is a kind of *run-length encoding (RLE)*.

6.2.1 Run-length encoding

RLE is based on the idea of compressing the similarities, where a sequence of identical values is replaced by its length: instead of reporting n sequential values v, it simply stores that v shows n time, $v[n]$. For example, the following set of integers is compressed in this way:

$$(4, 6, 2, 2, 2, 2, 2, 8, 3, 3, 7, 7, 7, 8) \rightarrow (4, 6, 2[5], 8, 3[2], 7[3], 8)$$

It is easy to notice that when sequences are very long, this encoding is very efficient.

It is also possible to omit the value v in the $v[n]$ notation if it is the only one chosen for the encoding; this happens in [1], where the only encoded value is obviously zero.

6.2.2 Zeros encoding

The following explanation of [1]'s solution is slightly adapted here in order to better introduce the new encodings proposed in this work: in this way, they will be easier to understand.

As the pruning percentage is usually very high, it is common to see long 0-sequences in the weights matrices. These matrices are seen in their vectorized form, where one row (or column) is attached to the end of the previous one. In this way they are a long array of values, which can of course be perfectly encoded. So, supposing A, B, C are different non-pruned elements, this matrix can be encoded by rows as

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A \\ 0 & 0 & 0 & 0 \\ B & C & 0 & 0 \end{pmatrix} \rightarrow (0, 0, 0, 0, 0, 0, 0, A, 0, 0, 0, 0, B, C, 0, 0, 0, 0) \rightarrow ([7], A, [4], B, C, [4])$$

which requires 6 elements instead of $5*4 = 30$ for the full matrix.

On an implementation point of view, *counters*, the zero-sequences lengths, can be represented on a lower number of bits with respect to the *weights*, which here will stand for *non-pruned weights*. However, using different sizes for counters and weights arises a problem: they cannot be identified in the stream of bits, as they have no more structure.

But knowing that on big, highly pruned matrices the probability to find two non-zero elements is very low - which is experimentally verified in the analysis done for this work - it is possible to decide a fixed schema to encode them and to avoid problems. Counters and weights will be alternated: so, if two weights are adjacent, the counter between them will be set to zero.

$$([n_1], w_1, [n_2], w_2, ..., [n_f], w_f)$$

The starting counter will be set to zero if the original matrix starts with a weight, or to the relative number of sequential zeros otherwise. It ends with a weight because the last sequence of zeros is useless and, knowing the dimension of the matrix, it can be implicit.

In the example above, the relative encoding would be the following, inserting a fake zeros sequence counter between B and C:

$$([7], A, [4], B, C, [4]) \rightarrow ([7], A, [4], B, [0], C)$$

Choosing N bits for the counters, it is possible to represent sequences long at most $2^N - 1$ elements. So, if some sequences are longer, a zero-weight can be inserted to bypass the problem. This can be useful when only few counters exceed $2^N - 1$ and adding a bit to all of them by increasing N would require more space than inserting these fake weights. In practice, a non-pruned element is considered a weight, even if it is zero, and it is inserted into the final encoding as if it was a w_k. Another motivation for applying this trick can be the same found in [1]'s work: they wanted word-aligned $([n_k], w_k)$ couples, and because weights were represented on twelve bits, they were obliged to choose $N = 4$, even if some sequences where longer than $2^4 - 1 = 15$.

In the example above, choosing to represent counters on two bits would result in the following encoding, remembering that the limit length is $2^2 - 1 = 3$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A \\ 0 & 0 & 0 & 0 \\ B & C & 0 & 0 \end{pmatrix} \rightarrow ([3], 0, [3], A, [3], 0, [0], B, [0], C)$$

Of course this is not the best choice for N, which in this case is instead $N = 3$.

Even if this encoding seems the best one and can be sustained by weights quantization and by the reduction of the number of bits required to represent them, this work will introduce an improvement to further compress matrices and also save operations thanks to the spiking phase results.

6.3 Proposed encodings and savings

The proposed optimizations are the outcome of a series of thoughts improved step by step, following the evolution that produced the clustering technique and then the spiking method. All these techniques were guided by the idea of normalizing the necessary weights to avoid performing redundant operations and to increase the similarity inside matrices.

6.3.1 Clustering advantages

The first attempt was done with clustering, which allows an intermediate weights compression and operations saving with respect to the previous works and the new approaches introduced by spiking.

6.3.1.1 Four-way encoding

Following the RLE idea, clusterized weights matrices allows a possible better compression. Using the same notation introduced in section 6.2, the following example shows how this is possible.

By introducing the $t[p]$ notation, where label t is 0 if p is a zeros sequence counter or $t = 1$ if p is a non-zero value, the following pruned matrix will have this encoding:

$$\begin{pmatrix} 0 & 0 & 0 & A \\ B & C & D & 0 \\ 0 & 0 & E & F \\ G & 0 & 0 & 0 \end{pmatrix} \rightarrow (0[3], 1[A], 1[B], 1[C], 1[D], 0[3], 1[E], 1[F], 1[G])$$

Supposing instead to have the same matrix after the clustering phase, so with the same amount of non-pruned weights in the same position, but most of them clustered into the two clustering centers λ_1 and λ_2, the resulting encoding would be the following. Two new rules are introduced for the $t[p]$ notation: t is λ_1 when p is a λ_1s sequence counter, and so for λ_2.

$$\begin{pmatrix} 0 & 0 & 0 & \lambda_1 \\ \lambda_1 & \lambda_1 & X & 0 \\ 0 & 0 & Y & \lambda_2 \\ \lambda_2 & 0 & 0 & 0 \end{pmatrix} \rightarrow (0[3], \lambda_1[3], 1[X], 0[3], 1[Y], \lambda_2[2])$$

The number of elements required to store the matrix in the first case is 9; in the second one, it is 6. Of course labels cannot be omitted as it happens for the zero-encoding: here, the repeated value must be specified to know which it is. But only four values are possible,

thus meaning that they can be encoded too. Just two bits are necessary to identify them, as they are only 0, λ_1, λ_2 and none of them. For example, they can be respectively encoded as 00, 01, 10 and 11.

As it is easy to see, this encoding can work reasonably well for not too much pruned matrices; otherwise, its ability to compress λ repetitions becomes useless. When a great number of weights get pruned away, the probability to find many adjacent λ values is very low, thus making the zeros-encoding less expensive thanks to its implicit labels.

Other ideas were tested: for example rewriting the matrix as a sum of other ones with a better possibility of encoding, as it is similarly proposed in the spiking-dedicated section below, but they were expensive anyway. However, all these ideas were re-adapted and used afterwards.

6.3.1.2 Operations savings

There is an operation performed by every neural network: the matrix-vector multiplication

$$\vec{y} = W\vec{x} \tag{6.1}$$

where W is the weights matrix, or one of them, and $\vec{x}$ is the input vector; collapsing the majority of weights into two opposite values can lower the amount of operations necessary to compute its result. Knowing that most of the weights are equal to the clustering centers $\pm\lambda$, it is possible to extract it from the matrix:

$$W = \lambda A$$

which means to perform an element-wise extraction of λ from all the elements w_k, indicated it in such a way:

$$w_k = \lambda \left(\frac{w_k}{\lambda}\right) = \lambda a_k \tag{6.2}$$

so, all the weights gets normalized by λ, which implies that most of the a_k elements of A are now equal to 1 or -1, and this is true when the starting element w_k is equal to $\pm\lambda$. For example, an extraction on a clusterized matrix can look like

$$\begin{pmatrix} 0 & 0 & 0 & 0.1 \\ 0.1 & 0.1 & 0.3 & 0 \\ 0 & 0 & -0.2 & -0.1 \\ -0.1 & -0.1 & 0 & 0 \end{pmatrix} = 0.1 \begin{pmatrix} 0 & 0 & 0 & 1.0 \\ 1.0 & 1.0 & 3.0 & 0 \\ 0 & 0 & -2.0 & -1.0 \\ -1.0 & -1.0 & 0 & 0 \end{pmatrix}$$

Thanks to this observation and by manipulating the order of multiplications in

$$\vec{y} = (\lambda A)\vec{x} \tag{6.3}$$

it is possible to drastically reduce the number of products to be computed. Writing the matrix-vector multiplication (6.3) in its explicit formulation for each element y_i of the result vector $\vec{y}$, the following equivalences are found:

$$y_i = \sum_{j=0}^{M-1} w_j x_j = \sum_{j=0}^{M-1} (\lambda a_{ij}) x_j = \sum_{j=0}^{M-1} a_{ij} (\lambda x_j) = \lambda \sum_{j=0}^{M-1} a_{ij} x_j$$

where x_j is the j-th element of the input vector $\vec{x}$ and w_j is the j-th element of the i-th row of W matrix. M is the number of elements in $\vec{x}$ and in W's rows. It demonstrates, obviously, the validity of this property:

$$(\lambda A)\vec{x} = A(\lambda \vec{x}) = \lambda(A\vec{x}) \tag{6.4}$$

So, looking for example at the first equality of the above (6.4) equation and calling $\vec{z} = \lambda\vec{x}$, the matrix-vector product (6.1) becomes:

$$\vec{y} = W\vec{x} = (\lambda A)\vec{x} = A(\lambda\vec{x}) = A\vec{z}$$

Once the M products represented by $\vec{z}$ are computed, only few of the $a_{ij}z_j$ are indeed true multiplications: as previously said, most of a_{ij} are now equal to 1 or -1. So, just M multiplications are needed instead of all the original $w_{ij}x_j$ ones where $w_{ij} = \pm\lambda$, which of course are usually much more than M: their quantity is equal to the number of clusterized elements, by definition. In other terms, once W has been off-line normalized into A, the number of product required is M plus the ones needed by the remaining non-clusterized weights. There are also some changes of signs to be performed instead of the fake multiplications where $a_{ij} = -1$, but there is also a way to avoid them; furthermore, even the M products can be avoided in some cases, as explained in section 6.4.

Therefore, in formulas, the matrix-vector multiplication is reduced to:

$$y_i = \sum_{j=0}^{M-1} a_{ij} z_j = \sum_{a_{ij}=1} z_j \; - \sum_{a_{ij}=-1} z_j \; + \sum_{a_{ij} \notin \{\pm 1, 0\}} a_{ij} z_j \tag{6.5}$$

As a side note, only the second member of the previous (6.4) equalities will be used for the purposes of this work: this is due to the fact that for LSTMs and GRUs more than

one W matrix is multiplied by $\vec{x}$, thus pre-calculating $\lambda\vec{x}$ saves a lot of operations, instead of post-multiplying each result vector by λ.

6.3.2 Spiking improvements

As it is easy to imagine, the spiking technique can eliminate all the clustering flaws, improving compression: weights can only assume three values: 0, σ or $-\sigma$. In a way similar to the one applied for the clusterized matrix with (6.2), a normalization can be applied to its elements by extracting σ:

$$W = \sigma S$$

Now, all s_k elements are 0, 1 or -1. From a practical point of view, this will only be useful for operations savings, but it also allows talking about pruned (0) or non-pruned (± 1) weights in an easier manner.

Also, it underlines an extremely important result of spiking: the following encodings will be independent from the number of bits chosen for weights representation. Once known the value of σ, only three kind of values have to be stored: thus, they can be substituted by three simple labels, for example on two bits. All these encodings are an attempt to take advantage of this detail in different ways.

After this overview, general spiking's improvements over operations savings will be introduced.

6.3.2.1 Matrix decomposition encoding

This proposed encoding needs a bit of previous formalization.

A sparse (pruned) matrix W whose elements can assume K possible values $v_1, v_2, ..., v_n$ apart from 0 can be split into a sum of K matrices

$$W = (L_1 + L_2 + ... + L_K) \tag{6.6}$$

where L_n is a matrix with only two possible values for elements, 0 or v_n. It is build setting to zero all the non-v_n elements of W: in this way, in a certain position, just one matrix will have a non-zero element, so that the resulting sum, which is performed position by position, will be again W. For example, if $K = 2$ and values are a and b, the following

matrix can be decomposed as

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ b & 0 & 0 & 0 \\ 0 & 0 & a & a \\ 0 & b & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & a & a \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 \end{pmatrix}$$

These two matrices can now be encoded as an array of alternated counters

$$[d_0][c_0][d_1][c_1]...[d_s][c_s]$$

in a sort of RLE manner. These counters implicitly and alternatively refer to a sequence of 0s or to a sequence of the other value. The only information to be saved is which is the starting value, and in this case it is 0 for both the matrices. So, discarding the last counter which is always implicit if matrix dimension is known - both if it refers to a (b) or 0, because its number can be derived - the example matrices can be encoded as

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & a & a \\ 0 & 0 & 0 & 0 \end{pmatrix} \rightarrow [10][2]$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 \end{pmatrix} \rightarrow [4][1][8][1]$$

Instead, the original matrix RLE code, which is conceptually equivalent to the previous counters approach, is:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ b & 0 & 0 & 0 \\ 0 & 0 & a & a \\ 0 & b & 0 & 0 \end{pmatrix} \rightarrow (0[4], b[1], 0[5], a[2], 0[1], b[1])$$

Here, the last counter could also be omitted because it is possible to choose a meaning for omission, too; of course, there is an high probability that a matrix ends with a zero, thus the best choice is to omit a zero counter if it is the last one.

So, the same amount of counters is required for the two encodings: in the decomposition case, 6 counters are needed, 2 for a-matrix and 4 for b-matrix; in the RLE case, always 6. Nevertheless, labels of the second case require space, too.

Generally speaking, the amount of counters strictly depends on the position of non-zero elements in the original matrix: considering that they are rarely adjacent, however, it is possible to state that decomposition encoding and original matrix RLE result in quite the same amount of elements. As an average case, it is possible to consider this basic pattern example where the last counter is omitted (matrix ends with zeros) and n and m are two indefinite length of zeros sequences:

$$(0[n], a[1], 0[m], b[1]) \leftrightarrow [n][1] \cup [n + 1 + m][1]$$

The two on the right are the encodings of the decomposition matrices. Always 4 values are required; but in general, for longer arrays, each a element adjacent to a b element reduces the number of elements required for RLE by one. As said, this eventuality is unlikely, and can be compensated by the amount of space required for labels.

Also, having two separate codes for the two matrices can be useful for an operations scheduler.

6.3.2.2 Two-bits encoding

Another approach is to notice that, for three values, just two bits are necessary as an encoding. So, instead of using an RLE, it is possible to simply encode all the weights on two bits, for example choosing 00, 10 and 11 for 0, 1 and -1 respectively. Normally, the fourth code, 01, would be wasted; but in this case, it can be used to produce a smarter compression thanks to the particular structure of the matrix. The fourth code can substitute long sequences of zeros, which can be represented by a counter. Counters will be stored separately, in order to keep the two-bits-structure of the resulting encoding regular, by using the fourth code like a marker or a pointer: the k-th 01 will point the k-th counter. Otherwise, counters can be inserted immediately after the fourth code, because they would be identified by its presence, like a prefix. The amount of bits will be the same: it is just a matter of decoder design ease. Again, the number N of bits required for the counters can be chosen properly.

Without this marker optimization, representing a sequence of zeros normally requires a number of bits equal to two times its length M. So, if $2M$ is greater than the length of the representation by means of the fourth code and the counter, which requires $N + 2$

bits, the sequence of 00 can be substituted profitably. The condition can be expressed as

$$2M > N + 2 \;\Rightarrow\; M > \frac{N}{2} + 1 \tag{6.7}$$

For example, the following normalized spiked matrix can be encoded without using the fourth code as

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow 00\ 00\ 00\ 11\ 11\ 00\ 00\ 00\ 00\ 00\ 10\ 00\ 10\ 00\ 00\ 00$$

but using the fourth code with $N = 3$, it is possible to shorten the sequence. Below, the first one is the main stream, while the second one is the counters stream. To make them better understandable, they are put together with a decimal and human-readable translation. The letter X will stand for the fourth code, *i.e.* the marker for sequences longer than $\frac{N}{2} + 1 = 2.5$.

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{array}{cccccccc} X & -1 & -1 & X & 1 & 0 & 1 & X \\ 01 & 11 & 11 & 01 & 10 & 00 & 10 & 01 \\ 011 & 101 & 011 & & & & & \\ (3) & (5) & (3) & & & & & \end{array}$$

The first marker X refers to the first counter, and so on. Calling M_k the length of the k-th substituted sequence, and P their total number, the second optimized encoding saves a number of bits with respect to the first one equal to

$$\sum_{k=0}^{P-1} (2M_k - (N+2)) = 2\sum_{k=0}^{P-1} M_k - P(N+2)$$

which is of course the sum of all the savings when condition (6.7) is true, *i.e.* the difference between the longer $2M_k$ bits representation and the shorter $N + 2$ one obtained by the use of the fourth code.

In this case, the second encoding has saved $2(3 + 5 + 3) - 3(3 + 2) = 7$ bits; the first encoding is 32 bits long, while the second one is only 25.

As a side note, counters are binary integers without signs, but the representation of 0, *i.e.* 000, would not make sense: if a counter exists, a sequence exists, thus its length cannot be zero. This remark allows the counters to be able to represent also the 2^N (8)

value with 000, while normally unsigned integers can not.

There is also another possible improvement. When, on large matrices, only few counters would require N bits instead of $N-1$ - if only few of them are greater than 2^{N-1} - or when slicing a sequence into smaller ones is convenient, it is possible to split the counters into several parts. For example, if in the example above $N = 2$ is chosen, the following encoding is found:

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{matrix} X & -1 & -1 & X & X & 1 & 0 & 1 & X \\ 01 & 11 & 11 & 01 & 01 & 10 & 00 & 10 & 01 \\ 11 & 00 & 01 & 11 & & & & & \\ (3) & (4) & (1) & (3) & & & & & \end{matrix}$$

The second sequence is split into two parts, so that two counters are obtained. But when the last sub-sequence representation with $N + 2$ bits - counters and marker - is longer than the one with a sequence of 00 - $2M^*$ bits if it is long M^* -, the above substitution condition (6.7) does not hold: it can be replaced back with 00s. So:

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{matrix} X & -1 & -1 & X & 0 & 1 & 0 & 1 & X \\ 01 & 11 & 11 & 01 & 00 & 10 & 00 & 10 & 01 \\ 11 & 00 & 11 & & & & & & \\ (3) & (4) & (3) & & & & & & \end{matrix}$$

which requires two bits less. It is a sort of remainder concept: if two times the remainder of the integer division between M and 2^N is lower than $N + 2$, thus if two times the remaining zeros to be represented is lower than the length of their representation with the fourth code, this back-substitution can be applied. The formula is:

$$2(M \ \% \ 2^N) < N + 2$$

In this case, the number of bits required for the encoding with $N = 2$ and the remainder is 24: the ones required for the encoding with $N = 3$ requires 1 bit more, 25. When matrices are larger, this effect can have an high contribution in the total number of bits required.

As always, the last element can be omitted if it refers to a sequence of zeros, thus making the bits required by this encoding equal to 20 for this example (also the last counter can be omitted).

6.3.2.3 One-bit encoding

As word or byte alignment can always be replicated by an intermediate decoder, its restrictions can be discarded, especially here where single codes are smaller than a word or even than a byte. In this way, further compression can be achieved, mixing the ideas of zeros encoding and two-bits encoding.

If zeros sequences are represented by counters, just two values have to be represented by a code: 1 and -1. So, they can be coded as binary 0 and 1, only with a single bit. This was not possible in two-bits encoding, because single zeros, markers and 1 and -1 had to be of the same length to be identified in the not regular stream, as their positions were variable. However, using a known, repeated structure like in zeros encoding, it is possible to recognize a weight-bit against a counter simply by its position. So, the basic repeated structure can be

$$[c_k], b_k$$

a counter on N bits followed by a weight on one single bit. In this way, the two redundant bits of the marker are avoided. If two weights are adjacent, the counter is set to zero, but this is not a huge problem for the rarity of the event, exactly as it was for zero encoding. As a side note, it means that the zero representation of unsigned integers is again necessary, so length can go normally up to $2^N - 1$. For example, the same matrix used for two-bits encoding becomes, with $N = 3$:

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{array}{cccccccccc} & & -1 & & -1 & & 1 & & 1 & \\ 011 & 1 & 000 & 1 & 101 & 0 & 001 & 0 & 011 & \\ (3) & & (0) & & (5) & & (1) & & (3) & \end{array}$$

Even if two weights are adjacent, which makes the counter waste very proportionately strong in this small example, this method still result in a better encoding, requiring 19 bits instead of the best result of the previous one, 24 (16 against 20 removing the last counter).

However, due to the same remark observed in zeros encoding, if a lower N is chosen and two counters are needed, the addition of a fake weight is not possible here, because all the 0 and 1 codes on a single bit are already occupied: but a notable consideration can be done, which also holds both for zeros encoding and possibly for two-bits encoding as a further optimization. Even if here elements must be identified by their position, a strict alternation is not required. Two subsequent counters can be recognized as such thanks to this hint: if a counter has a follower, it means that its value is set to the greatest one

possible, $2^N - 1$. However, the reversed implication is not true: if a counter is set to $2^N - 1$, $2^N - 1$ can be the exact length of the sequence. To overcome this little flaw, a fake counter set to zero will be inserted afterwards, and it is not a significant waste: maybe, none of the lengths is perfectly equal to $2^N - 1$ and generally speaking their amount will be derisory. In this way, the decoder can perfectly identify a following counter without confusing it with a weight. Therefore, the basic repeated structure becomes

$$[c_k]^+, b_k$$

where the + sign stands for "one or more", as it is for regular expressions. So, for example, if $N = 2$ is chosen to encode the same matrix without the last 0s sequence:

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{array}{cccccccccccc} & & & & -1 & & -1 & & & 1 & & 1 \\ 11 & 00 & & 1 & 00 & 1 & 11 & 10 & 0 & 10 & 0 & \\ (3) & (0) & & & (0) & & (3) & (2) & & (2) & & \end{array}$$

this encoding requires again only 16 bits, even if it is applied in the worst case (all the two improbable situations appear). For example, if the first -1 was in the previous position, $N = 3$ would led to 17 bits, while $N = 2$ only 14.

$$\begin{pmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{array}{cccccccccccc} 010 & 1 & 001 & 1 & 101 & & 0 & 010 & 0 & (N = 3) \\ 10 & 1 & 01 & 1 & 11 & 10 & 0 & 10 & 0 & (N = 2) \end{array}$$

Also, two-bits encoding would be worse, even if $N = 3$ would be better than $N = 2$, because just one marker would be used: it would require 21 bits, without the last counter. For larger matrices, this difference with two-bits encoding will be greater, as markers are avoided and spiked weights are represented on a single bit.

6.3.2.4 Operations savings

As seen for the clustering technique, it is possible to operate some manipulations on operations order to perform a significantly lower number of computations. Spiking is far more efficient in doing so: it works even for highly pruned networks and eliminates all the non-clustered weights left by its ancestor.

Following the same procedure shown for clustering with equation (6.3), the matrix-vector multiplication $W\vec{x}$ can be rewritten as

$$\vec{y} = W\vec{x} = (\sigma S)\vec{x} = S(\sigma\vec{x}) = S\vec{z}$$

by extracting σ from W and calling $\sigma\vec{x} = \vec{z}$. However, as the S matrix contains only elements equal to 0, 1 or -1, a great result can be achieved: it is possible to eliminate the third contribution in the equation (6.5), as there are no more residual weights. It becomes:

$$y_i = \sum_{j=0}^{M-1} s_{ij} z_j = \sum_{s_{ij}=1} z_j \; - \sum_{s_{ij}=-1} z_j \tag{6.8}$$

In other words, a fundamental result has been reached: the matrix-vector multiplication of (6.1) has been reduced to a very limited number of sums, also considering the possibility to avoid the $\sigma\vec{x}$ product introduced in section 6.4. Furthermore, because of the grouping shown in Chapter 5, a single $\sigma\vec{x}$ can be used for multiple matrix-vector multiplications, such as in the case of the four forward matrices of LSTM. For a quantitative estimation of savings, see section 6.4.

Similarly, it is possible to see this result using (6.6) decomposition and extracting $\pm\sigma$:

$$\vec{y} = W\vec{x} = (P+Q)\vec{x} = (\sigma B + (-\sigma)D)\vec{x} = \sigma B\vec{x} - \sigma D\vec{x} = B\vec{z} - D\vec{z}$$

where B and D are binary matrices obtained by P and Q that simply tell which elements of z must be picked and added together to obtain the elements of $\vec{y}$. In this sense, separated codes for matrices can be useful for an operation decoder and scheduler: two accumulators can add this two separated results, and then a subtractor can give the final result, so that no change of sign is necessary, and no additional operation at all, too. The number of operations in formula (6.8) is indeed $(p-1)+(n-1)$ additions plus 1 subtraction, where p is the number of elements equal to 1 and n is the number of elements equal to -1, so

$$(p-1)+(n-1)+1 = p+n-1$$

operations; by adding together all the elements instead, after a change of sign, the number of operations would have been $p+n-1$ additions plus one change of sign for each of them, so $2(p+n-1)$. Even if the change of sign is operated only after the two separate $p-1$ and $n-1$ accumulations, so $p+n-2$ plus 1 sign change ($p+n-1$ operations), there is still an addition to be performed, making the first scheduling, represented by (6.8), the best one. Of course, this additional operation can be preferred in hardware design choices

if a subtractor is undesired, like a sort of trade-off.

This is the core of this work: up to now, only the properties of number 0 were exploited in previous works to avoid multiplications after the pruning method. However, multiplication has two other particular numbers, 1 and −1:

6.4 Final results and comparisons

6.4.1 Encodings

All these encodings were tested on different pruning percentages: of course, higher percentages imply higher compressions. The final size of the LSTM network will be compared to the initial one, where no spiking or pruning has been applied: of course, the compression factor is computed as the inverse of the final size percentage with respect to the original one. As shown in Figures 6.1, 6.2 and 6.3, the one-bit encoding is always the best one.

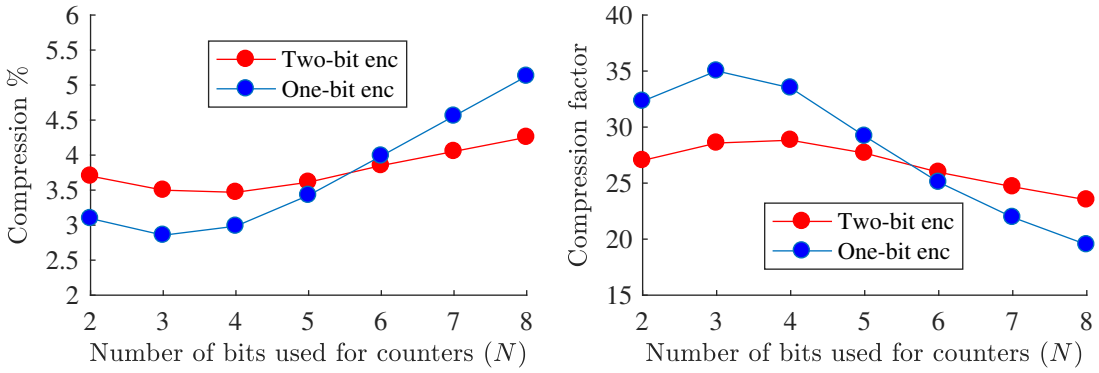


Figure 6.1: Pruned weights: 82%.

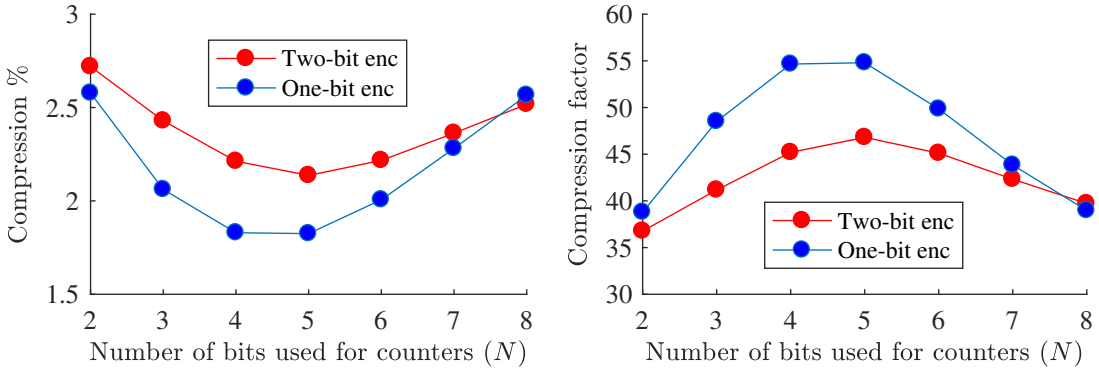


Figure 6.2: Pruned weights: 91%.

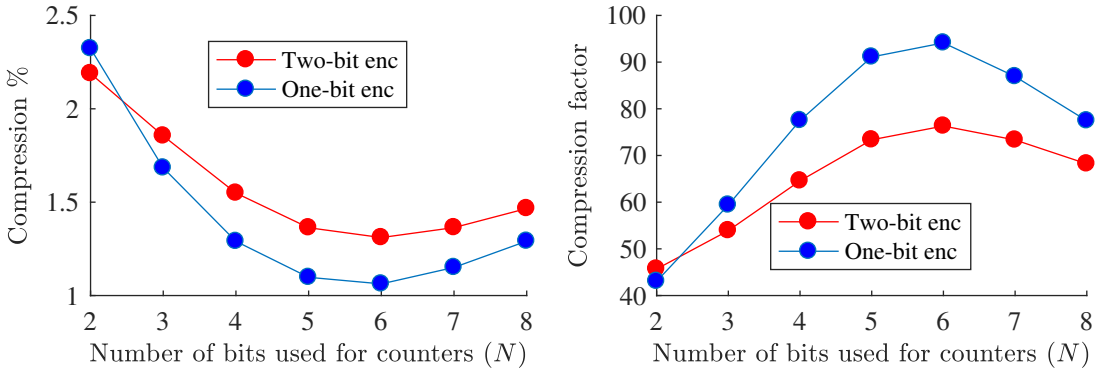


Figure 6.3: Pruned weights: 95%.

However, when the pruning percentage reaches very high values, a more simple and common *Compressed row storage (CSR)* could be better. Reading the matrix M row by row, this method saves for each non-pruned element its value and the index in M of its column: it is a group v of parallel arrays of N_s elements each, where N_s is the number of non-zero weights. Separately, it stores the index in v of the first non-pruned element of the row, for each row of M, in order to understand when to change line while decoding v: another array r, whose dimension is $R \leq n$, the number of rows with at least one non-pruned weight, where n is their total number in M. This simple method can achieve good performances when the sparsity of the matrix is very high: it needs just $(2N_s) + (R + 1)$ storage locations, where $+1$ stands for the element that permits the acknowledgment of the end of the structures (an index equal to $R + 1$ as last element of r). To confront it with the proposed methods, a single LSTM matrix is taken, in order to have its exact pruning percentage: as shown in Figure 3.6, matrix density is different for each of them. In general, these two methods performed better (Figure 6.4):

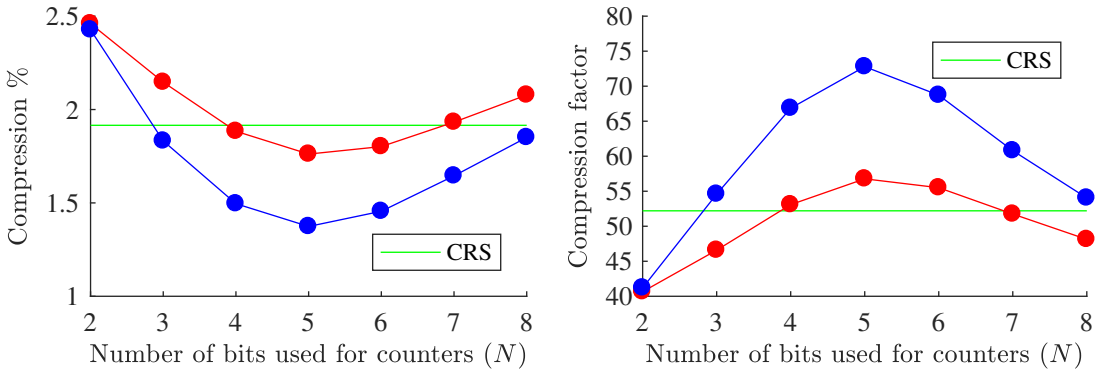


Figure 6.4: The two proposed encodings perform better than CRS.

But, in a very extreme case, when the pruning percentage is 99.78% and the remaining 36 weights are on $R = 31$ different rows instead of 36, CRS almost reaches one-bit encoding (Figure 6.5). After this moment, for example because $R < 31$, CRS performs better. However, this example shows that the compression factor is higher than 1000× with only a 4% of accuracy loss (this is type D, seed 205).

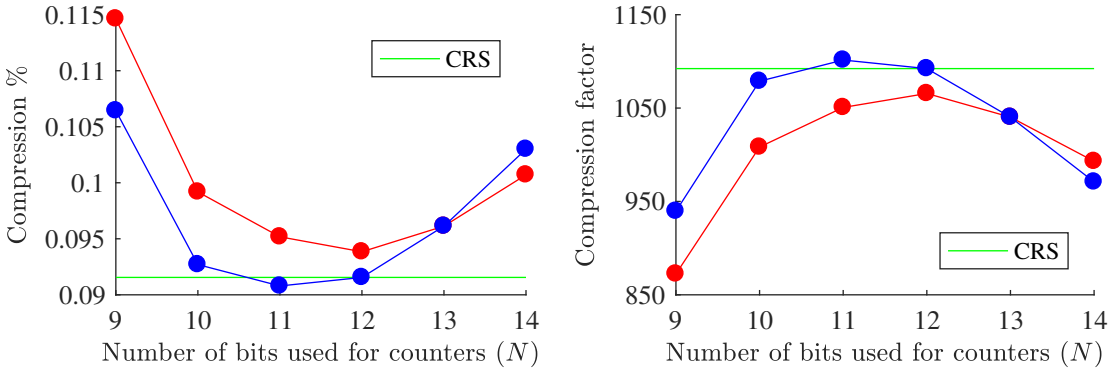


Figure 6.5: Extreme case: limit of one-bit encoding against CRS.

The green value is computed with the formula above, but looking at the bits for a precise comparison: the values of the elements in v are stored on a single bit (−1 and 1), while their column index is on 7 bits (128 rows); elements in r are represented on $\lceil log_2(N_s - 1) \rceil$ bits, the minimum to represent $N_s - 1$, which is the maximum index to point an element in v. So, with this formula:

$$N_s + 7N_s + \lceil log_2(N_s - 1) \rceil (R + 1)$$

6.4.2 Savings

The multiplications savings can be quantified by the ratio between the total required number of products after a technique application and the number of initial ones. Originally, for the LSTM case, eight matrix-vector multiplications must be computed, thus $8n^2$ products, as matrices are $n \times n$: indeed, one product for each weight. Calling $\mathcal{X}$ the number of required multiplications, the actual saving can be expressed with a number between 0 (no saving) and 1 (total saving) through this formula, which compares the saved operations $8n^2 - \mathcal{X}$ to the initial ones:

$$\phi = \frac{8n^2 - \mathcal{X}}{8n^2} = 1 - \frac{\mathcal{X}}{8n^2}$$

If p is the pruning percentage, then $8pn^2$ is the number of zero elements, and so the number of saved multiplications. Therefore, the total number $\mathcal{X}_p$ of products after the

pruning method is:

$$\mathcal{X}_p = 8n^2 - 8pn^2 = 8(1-p)n^2$$

But calling q the percentage of clusterized weights with respect to the number of non-pruned weights N_s, the relative method can achieve a better result. By extracting λ, qN_s multiplications can be saved by simply computing $\lambda\vec{x}$ for the forward matrices and $\lambda\vec{h}$ for the recurrent matrices: the same input $\vec{x}$ is given to the forward group and the same previous output $\vec{h}$ is given to the recurrent group. So, they are $2n$ products. The remaining $(1-q)N_s$ weights have to be multiplied anyway, because they are not normalized to 1 or -1. Therefore, the total number of multiplications is $(1-q)N_s + 2n$, or equivalently:

$$\mathcal{X}_c = 8(1-q)(1-p)n^2 + 2n$$

However, the spiking method can eliminate the first term contribution, because all the weights are spiked: if we see q as the percentage of collapsed weights with respect to N_s, then $q = 1 \Rightarrow (1-q) = 0$. Indeed, if the two sets of recurrent and forward matrices U_* and W_* are grouped to find σ_r and σ_f respectively, the spiking technique only requires the $\sigma_f\vec{x}$ and $\sigma_r\vec{h}$ multiplications, thus a number of operations equal to

$$\mathcal{X}_s = 2n$$

Therefore, generally speaking, $O(n)$ products against $O(n^2)$: linear against quadratic. When matrices are very large, this effect can be very powerful; also, it does not depend on the pruned percentage, so that savings are very high even p is possibly very low, when for example compression is not so important but power consumption and time are. Simply with eight 128×128 matrix, the saving is

$$\phi_s = 1 - \frac{2n}{8n^2} = 1 - \frac{1}{4n} = 0.998 \rightarrow 99.8\%$$

Nevertheless, if the input vector $\vec{x}$ is given to the network already normalized by σ_f, then of course its n products can be saved. Even if this idea can seem strange, in many contexts it is not: for example, the input of a network can be the output of a previous process or even another network; it can be arbitrary manipulated for many reasons to adapt it; it can be simply arbitrary. This is the case of the tested network: the input of the LSTM unit is a vector whose components are decided during the training phase. So, they can be normalized by σ_f, eliminating the necessity of an on-line multiplication at each iteration.

The resulting number of operations is given by the only $\sigma_r \vec{h}$ product, and the saving is:

$$\mathcal{X}_s^{'} = n$$

$$\phi_s' = 1 - \frac{n}{8n^2} = 1 - \frac{1}{8n} = 0.999 \rightarrow 99.9\%$$

Furthermore, also the $\sigma_r \vec{h}$ product can be avoided in some cases. If a highly pruned network is chosen, the recurrent connections are set to 0: so, all U matrices are void, which implies that no recurrent products must be computed. Also, looking at these result, it can be possible to arbitrary drop them, forcing the pruning algorithm to eliminate them without touching W_c, because it is the fundamental matrix, as shown in Chapter 3. However, generally speaking, all recurrent products are automatically discarded if the target network is feed-forward: either if the model was altered or if these methods are simply applied to feed-forward networks. In this case:

$$\mathcal{X}_s^* = 0$$

$$\phi_s' = 1 \rightarrow 100\%$$

In this way, spiking can eliminate all the multiplications: while designing a dedicated hardware device, arrays of multipliers can be totally avoided or enormously simplified if just n products are necessary.

Chapter 7

Implementation of a general tool

7.1 Starting point

A general software tool was developed to train a model and apply all the techniques introduced above. It was written in Python using Theano [26] [27], a library for compiling and optimizing complex mathematical functions in order to make their execution very fast with high parallelization on CPUs. Theano can also work very well for *GPGPU*, which stands for *General-Purpose computing on Graphics Processing Units*, a usage of GPUs which is not related to graphics but simply to exploit its massively-parallel architecture. This library allows the code to take advantage of GPUs' computational power in a transparent way, without a change on its lines; this will be useful for future works where this tool could be used on GPUs, because, for the little studied network, CPU computing was enough. Furthermore, Theano offers symbolic differentiation capabilities, which are very helpful for backpropagation and so for neural network modeling.

Because SGD optimizers are nowadays well studied and widely used, many are their implementations. For example, the case-of-study network was written by [15] using Theano; also, some code to train it was provided, with the standard SGD procedures. This was very useful, because differently from other higher level libraries where *AdaDelta* and other optimizers have already been wrapped in function calls, here all the details were visible and so they could be manipulated and extended to quickly implement the new techniques without starting from scratch.

All the experiments were run on HACTAR, one of the two academic computing systems of Politecnico di Torino [28]. HACTAR is an InfiniBand cluster of 24 nodes which can perform 9 TFLOPS; each of the nodes has 24 Intel XEON v3 core and 128GB of RAM.

7.2 Description

The developed tool offers a flexible way for optimizing a function with respect to some variables, which is tipically a network. It also applies one or more of the proposed methods, showing the resulting matrices.

This tool offers an adaptable and parametrized Python function, whose arguments allow the specification of some behaviors and can be used to communicate the name of a file in which further details are listed. These arguments will be explained below, together with the most important ones of the configuration file.

```
train_compress(
    seed,                       # [Integer]
    max_kilobytes,              # [Integer]
    prepare_data,               # [Function] x, mask, y = prepare_data(x, y)
    train, valid, test,         # [Tuple of arrays] (array_of_inputs, labels)
    build_net_model,            # [Function] build_net_model(options)
    options_training,           # [Dictionary]
    options_pruning,            # [Dictionary]
    options_clustering,         # [Dictionary]
    options_spiking,            # [Dictionary] .
    filters,                    # [Array]
    notfilters=None,            # [Array]
    report_file='report.txt',   # [String]
    configuration_file=None     # [String]
)
```

`seed` is the integer received by the function to initialize the random numbers generator for SGD: in this way, passing the same value will generate the same results. This is useful to exactly repeat an experiment and obtain its precise results.

`max_kilobytes` is a constraint used by the function to choose the model with the best accuracy among the pruning steps: the best one that can be also compressed at least to this size with spiking and one-bit encoding will be picked.

`prepare_data` for recurrent network, this function can adapt data of mini-batches in order to have them on the proper length through the usage of a $0-1$ mask; this is useful for Theano to make the management of different length sequences simpler and quicker.

`train, valid, test` are the datasets with the proper label that the network will try to obtain; of course, just the training set will be used for SGD, while validation set is used to avoid overfitting and test set is used only for final evaluation. However, just to see test accuracy evolution during the training phase for tuning purposes, it will be used to print also the test error with a proper frequency.

`build_net_model` is a Python function called before training, pruning, clustering and spiking to build the model of the network in a Theano format. It specifies and returns the format of inputs and outputs, together with the Theano cost function to minimize and other details. Also, it must return a dictionary of weights as Theano variables so that each matrix (or group of matrices) is labeled with a string name; here, biases are listed, too.

`options_training, options_pruning, options_clustering, options_spiking` are the arguments received by `build_net_model(options)` function, one for each phase. As

it is a totally user-written function, these parameters can be useful to specify different behaviors for each model. For example, they can be used to specify the use of dropout for training, but not for the other methods.

`filters` is an array that specifies which of the matrices will be the target of the compression techniques, by listing their names in the previously introduced dictionary.

`notfilters` is an optional array that specifies which of the parameters (matrices, biases and so on) in the dictionary must also be considered into the `max_kilobyes` account: they are the non-prunable weights which must be stored respecting the memory constraint.

`report_file` is the name of the file where reports will be written. Aside from the global and summarizing one, different files will be produced for each technique with further details. Their name will have this string as a suffix.

`configuration_file` is the name of the file where all optional configurations are listed in a `key<space>value` format. They have otherwise default values.

7.2.1 Configurations

Many are the configurations options, so only the most important ones within the following list will be briefly explained to introduce the significant aspects of the developed tool.

```
patience 10                     # [Integer]
disp_freq 10                    # [Integer]
lrate 0.0001                    # [Float]
valid_freq 370                  # [Integer]
save_freq 1110                  # [Integer]
batch_size 16                   # [Integer]
valid_batch_size 16             # [Integer]
training_max_epochs 200         # [Integer]
pruning_max_epochs 20           # [Integer]
clustering_max_epochs 30        # [Integer]
spiking_max_epochs 15           # [Integer]
threshold_just_clustering None  # [Float]

pruning_steps 11                # [Integer]
clustering_steps 11             # [Integer]
final_radius MAX                # [Integer or String]
slow_pruning 0.005              # [Float]
slow_clustering 0.05            # [Float]
slow_spiking 0.00001            # [Float]
choose_just_accuracy true       # [Boolean]
trained_model None              # [String]
pruned_model None               # [String]
threshold_shift 0.0             # [Float]
do_clustering false             # [Boolean]
do_spiking_foreach_pruning true # [Boolean]
```

`pruning_steps`, `clustering_steps` are the number of steps S used for pruning and clustering.

`final_radius` is the last ρ_S used to clusterize all the weights. Normally, it is `'MAX'`, as it is set to λ.

`slow_pruning, slow_clustering, slow_spiking` are the γ slowing factor introduced with the various methods.

`choose_just_accuracy` is used to pick the best pruned model among all the steps: if it is true, the model with the highest accuracy respecting the memory constraint will be chosen; otherwise, the model with the best compromise between accuracy and required size is picked. This compromise is a simple quadratic sum of the two percentages.

`trained_model` is the name of a previously trained model saved in a special format, `npz`, which is the same in which the tool saves all of them, so that the pruning phase will directly start without the launching of a training stage.

`pruned_model` is the name of a pruned model which will be clustered or spiked. The pruning and the training phases are skipped.

`threshold_shift` is the δ parameter introduced in Chapter 4 to shift the clustering center λ from the threshold τ.

`do_clustering` tells whether to perform clustering or spiking. Normally, this option is set to false.

`do_spiking_foreach_pruning`, if false, the tool performs spiking only on the best pruned model among all the steps; otherwise, the spiking technique will be applied separately for each of them.

7.2.2 Reports

Among the various reports of the tool, the main one lists all the accuracies found by the various phases. Respecting the format of the starting point script, it prints the error, which is simply one minus the accuracy. This is an example:

```
<<<<<<REPORT FILE>>>>>>

TRAINING
Iter: 0 Pruning: 0.0 Error: 0.1118


PRUNING
Iter: 1 Pruning: 0.0911865234375 Error: 0.11232
Iter: 2 Pruning: 0.180961608887 Error: 0.11124
Iter: 3 Pruning: 0.2734375 Error: 0.11112
Iter: 4 Pruning: 0.36450958252 Error: 0.11156
Iter: 5 Pruning: 0.455215454102 Error: 0.11204
```

```
Iter: 6 Pruning: 0.545562744141 Error: 0.11268
Iter: 7 Pruning: 0.637092590332 Error: 0.11196
Iter: 8 Pruning: 0.726356506348 Error: 0.11272
Iter: 9 Pruning: 0.817710876465 Error: 0.11416
Iter: 10 Pruning: 0.908699035645 Error: 0.11616
Iter: 11 Pruning: 0.999687194824 Error: 0.15732

BEST ACCURACY (under pruning constraint) --- Iter: 9 Pruning: 0.817710876465
      Error: 0.11416
BEST COMPROMISE --- Iter: 10 Pruning: 0.908699035645 Error: 0.11616


SPIKING
Error: 0.111
```

The other specific reports lists for example the value of the spiking center, the number of clustered weights and so on. The tool also prints the time required by each step: this times depends on many factors related partially to the HACTAR cluster and also to the possible early stop if accuracy does not improve for many epochs. To have an idea of the order of magnitude for type D, these time are in the order of $4*10^4 s$ for each step. Considering all the 11 steps of pruning, plus training and a single spiking, times of execution are around 6 or 7 days.

7.3 Pseudocodes

7.3.1 Dichotomic search

The dichotomic search is used to find the right τ for each pruning step. A tolerance of 0.001 is used to obtain the desired pruning percentage in Pseudocode 7.1.

Pseudocode 7.1: Dichotomic search

```
threshold = INITIAL_VALUE
maxthreshold = MAX_VALUE
minthreshold = MIN_VALUE
perc_notfound = True
search_time = 0
while(perc_notfound and search_time<MAX_ITERATIONS)
    pruned_percentage = count_weights_greater_than(threshold)
    perc_notfound = abs(pruned_percentage - DESIRED_PERC)>TOLERANCE
    if(perc_notfound)
        search_time += 1
        if(pruned_percentage>pruning_perc)
            maxthreshold = threshold
        else
            minthreshold = threshold
        threshold = (maxthreshold + minthreshold)/2.0
```

7.3.2 Encodings

Pseudocodes 7.2 and 7.3 show how two-bits encoding and one-bits encoding can be implemented. There is also a formula to compute the total number of required bits. As it is possible to see here, two-bits encoding is far more complex than one-bit encoding.

Pseudocode 7.2: Two-bits encoding

```
flag_seq = false       # Utility flag for printing
two_enc_w = 0          # Total number of '10' or '10' weights
two_enc_markers = 0    # Total number of '01' markers
two_enc_singles = 0    # Total number of '00' single zeros
two_enc_counters = 0   # Total number of counters
m_zeros = 0            # Number of adjacent zeros in a sequence
N = TWO_ENC_N          # Number of bits used for counters
max_value = 2^N        # Max representable value on a single counter

for(i from 0 to length(vectorized_matrix)-1)
    if(vectorized_matrix(i) == 0)
        m_zeros = m_zeros + 1
    else
        if(m_zeros <= (N/2 + 1))           # Using single values is better
            print_singles(m_zeros)         # Print multiple '00'
            two_enc_singles = two_enc_singles + m_zeros
        else
            print_marker()                 # Optimization: always a single marker
            two_enc_markers = two_enc_markers + 1
            if m_zeros < max_value         # Single counter
                print_counter(m_zeros, N)  # Binary value of the counter on N bits
                two_enc_counters = two_enc_counters + 1
            else
                remainder = mod(m_zeros, max_value)
                n_counters = ceil(m_zeros/max_value)     # Required counters
                if(remainder!=0 and 2*remainder<N)       # N+2 if n markers
                    n_counters = n_counters - 1          # Replace last counter
                    m_zeros = m_zeros - remainder        #   with '00' sequence
                    flag_seq = true
                print_counters(m_zeros, N, n_counters)   # On n counters
                if(remainder == 0)                       # Fake counter
                    print_counter(0, N)
                    n_counters = n_counters + 1
                else if(flag_seq)
                    print_singles(remainder)
                    two_enc_singles = two_enc_singles + remainder
                    flag_seq = false
                two_enc_counters = two_enc_counters + n_counters
        m_zeros = 0
        print_weight(vectorized_matrix(i))
        two_enc_w = two_enc_w + 1
# Possible last zero sequence can be omitted
total_bits = two_enc_counters*N + (two_enc_singles+two_enc_markers+two_enc_w)*2
```

Pseudocode 7.3: One-bit encoding

```
one_enc_w = 0           # Total number of one-bit weights
one_enc_z = 0           # Total number of counters
m_zeros = 0             # Number of adjacent zeros in a sequence
N = ONE_ENC_N           # Number of bits used for counters
max_value = 2^N-1       # Max representable value on a single counter

for(i from 0 to length(vectorized_matrix)-1)
    if(vectorized_matrix(i) == 0)
        m_zeros = m_zeros + 1
    else
        if(m_zeros < max_value)       # True even if the first element is non-zero
            print_counter(m_zeros, N) # Binary value of the counter on N bits
            one_enc_z = one_enc_z + 1
        else
            n_counters = ceil(m_zeros/max_value)   # Required number of counters
            print_counters(m_zeros, N, n_counters) # Print on multiple counters
            if(mod(m_zeros,max_value) == 0)        # Add fake counter
                print_counter(0, N)
                n_counters = n_counters + 1
            one_enc_z = one_enc_z + n_counters
        m_zeros = 0
        print_weight(vectorized_matrix(i))
        one_enc_w = one_enc_w + 1
# Possible last zero sequence can be omitted

total_bits = one_enc_z*N + one_enc_w*1;
```

Conclusions

State of the art comparisons

The best accuracy reached in literature on IMDB movie review dataset with this LSTM architecture is 88.89% [16]; other approaches are possible, and the winner of a Kaggle competition obtained an under ROC curve area of 0.9925 [29]. The aim of this work was not to reach a better accuracy, but to keep it stable with a great reduction of model expressiveness, and it has been fully achieved; furthermore, in a particular case, an accuracy of 89.9% has been reached.

The other main goal was to better compress the network and save operations, which was again accomplished. The resulting model of LSTM weights matrices is $50 - 100$ times smaller than the original one with the proposed one-bit encoding: its exact value depends on the pruning percentage and the affordable accuracy loss. It can also reach more than $1000\times$ if 4% loss is acceptable. Other works obtained a value equal to 20[1] or 32[7] for this factor and similar methods.

Final comment

Differently from most of the quantization works, these techniques can compress the network as if it was quantized and can save operations as if it was truly binarized even if additions and values keep their precision untouched, whatever representation is chosen: binarization is only virtualized by the extraction of full-precision σ from the matrices. Of course, these operations savings can be useful for software models, too: the new local minimum found by these methods can transform a matrix-vector multiplication into a limited number of sums, which is a great result even for reducing the temporal and spatial complexity of a software operation.

Comparing the two last and proposed methods, clustering technique led to worse result probably because λ was arbitrary and not customized according to the weights. Spiking method, instead, finds the best σ by itself and it is far more powerful: it collapses all the non-pruned weights together in two values, using a projection on the new constrained solution space, and can achieve better compression.

Furthermore, pruning and spiking work well together: after the training phase assigns a role to each connection, which can be positive or negative, the pruning phase finds the right

reduced *structure* of all the connections, while the spiking phase search for their optimal *value*. They can achieve very good compressions thanks to the three-peaks distribution of the final weights.

Future works

These techniques could be applied to other networks - like feed-forward ones - and for other tasks, not only for sentiment analysis, to see what happens. However, thanks to the 99.9% pruning, the LSTM model has been transformed into a non-recurrent one, because all recursive connections have been cut, suggesting inevitably the effectiveness of these methods also for feed-forward networks.

The spiking center update can be further improved by modifying the learning rate formula, to find the better expression to rescale the partial derivatives. Here, the *AdaDelta*'s one has been used, but it is possible that another type of formula can achieve better results if customized for the semi-bisectors solution space.

Also, instead of trying to approximate a local minimum with a point on the constrained space, it could be possible to distort the initial space A into another one, D, in order to directly move the optimum on one of its semi-bisectors.

To improve the compression, as it depends on the number of adjacent non-pruned elements, an evolutionary algorithm can be applied to swap two or more rows and two or more columns of the matrix in order to regularize its structure, because each change can be compensated by simply swapping some elements of the input or the output vector of the product, without loss.

As a side note, other ideas were discarded during the development of this work; in future, they can be further studied. For example, it can be possible to train a smaller network to perfectly reproduce a section k of the encoding bits stream when index k is given as an input, for each possible k: this will transform memory in computation, in order to save space while paying power consumptions.

Bibliography

[1] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, W. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA", FPGA, 2017.

[2] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, D. Marr, "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC", 26th International Conference on Field Programmable Logic and Applications (FPL), 2016.

[3] Y. Guan, Z. Yuan, G. Sun, J. Cong, "FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks", ASP-DAC, 2017.

[4] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", EMNLP, 2014.

[5] S. Han, J. Pool, J. Tran, W. J. Dally, Learning both weights and connections for efficient neural networks, NIPS, 2015.

[6] S. Han, H. Mao, W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding", ICLR, 2016.

[7] M. Rastegari, V. Ordonez, J. Redmon, A.Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks", ECCV, 2016.

[8] J. Weston, S. Chopra, A. Bordes, "Memory Networks", ICLR, 2015.

[9] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, "Long-term Recurrent Convolutional Networks for Visual Recognition and Description", CVPR, 2015.

[10] N. Grabaskas, D. Si, "Anomaly Detection from Kepler Satellite Time-Series Data", Machine Learning and Data Mining in Pattern Recognition, pp.220-232, 2017.

[11] T. Charnock, A. Moss, "Deep recurrent neural networks for supernovae classification", The Astrophysical Journal Letters, 2017.

[12] Y. Bengio, P. Simard, P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult", IEEE Transactions on Neural Networks, vol. 5, no. 2, pp. 157-166, 1994.

[13] S. Hochreiter, J. Schmidhuber, "Long Short-Term Memory", Neural Computation. 9 (8): 1735–1780, 1997.

[14] F. A. Gers, J. Schmidhuber, F. Cummins, "Learning to forget: Continual prediction with LSTM". Neural computation, 12(10), pp. 2451-2471, 200.

[15] P. L. Carrier, K. Cho, LSTM Networks for Sentiment Analysis, Deeplearning, `http://deeplearning.net/tutorial/lstm.html`

[16] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, C. Potts, "Learning Word Vectors for Sentiment Analysis", 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 2011.

[17] IMDB large movie review dataset for sentiment analysis, `http://ai.stanford.edu/~amaas/data/sentiment/`

[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR, 2014.

[19] S. Ruder, "An overview of gradient descent optimization algorithms", arXiv preprint arXiv:1609.04747, 2016.

[20] B. Hassibi, D.G. Stork, G.J. Wolff: "Optimal Brain Surgeon and general network pruning", IEEE International Conference on Neural Networks, 1993.

[21] Y. LeCun, J.S. Denker, S.A. Solla, R.E. Howard, L.D. Jackel, "Optimal brain damage", NIPS, 1989.

[22] S. Han, J. Pool, J. Tran, W. J. Dally, "A pruning based method to learn both weights and connections for LSTM", NIPS, 2015.

[23] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, W. J. Dally, "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks", NIPS, 2017.

[24] P. Merolla, R. Appuswamy, J. Arthur, S. K. Esser, D. Modha, "Deep neural networks are robust to weight binarization and other non-linear distortions", arXiv preprint arXiv:1606.01981, 2016.

[25] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1", arXiv:1602.02830, 2016.

[26] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, Y. Bengio, "Theano: new features and speed improvements", NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2012

[27] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, "Theano: a CPU and GPU math expression compiler", Proc. of the Python for Scientific Computing Conference (SciPy), 2010.

[28] Computational resources were provided by HPC@POLITO, a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino (`http://www.hpc.polito.it`)

[29] Bag of Words meets Bags of Popcorn, Kaggle Competition on IMDB dataset, `https://www.kaggle.com/c/word2vec-nlp-tutorial`