UNIVERSITÀ DEGLI STUDI DI TRIESTE

UNIVERSITÀ DEGLI STUDI DI TRIESTE

XXXV CICLO DEL DOTTORATO DI RICERCA IN

INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

PO FRIULI VENEZIA GIULIA - FONDO SOCIALE EUROPEO 2014/2020

An Analysis of Pruning in Deep Neural Networks: Experimental Investigations and Applications

Settore scientifico-disciplinare: **ING-INF/05**

DOTTORANDO / A
MARCO ZULLICH

COORDINATORE
PROF. ALBERTO TESSAROLO

SUPERVISORE DI TESI
PROF. FELICE ANDREA PELLEGRINO

CO-SUPERVISORE DI TESI
PROF. ERIC MEDVET

ANNO ACCADEMICO 2021/2022

An Analysis of Pruning in Deep Neural Networks: Experimental Investigations and Applications

a Ph.D. dissertation

by

Marco Zullich
M.Sc. in Data Science and Scientific Computing, University of Trieste, Italy

Marco Zullich, M.Sc.

An Analysis of Pruning in Deep Neural Networks: Experimental Investigations and Applications

a thesis submitted for the title of **Ph.D. in Industrial and Information Engineering**

Dissertation date: February 22rd, 2023
Supervisor: Prof. Felice Andrea Pellegrino
Co-supervisor: Prof. Eric Medvet

University of Trieste
Department of Engineering and Architecture
Ph.D. course in *Industrial and Information Engineering*
Via A. Valerio, 6/1 - 34127 Trieste, Italy

Life will only change when you become more committed to your dreams than you are to your comfort zone.

—Billy Cox

Acknowledgements

My most sincere gratitude goes to all the people who supported me during this adventure, for their invaluable assistance and encouragement.
To Franz and Yvonne for backing my crazy idea of resuming my master's studies after six years of working.
To my parents Gabriella and Roberto for ever being helpful and supportive.
To my supervisors Felice Andrea Pellegrino and Eric Medvet, that guided me with expert hand throughout this three-year long journey.
To Luca Bortolussi for his consistent commitment to creating and running a competitive AI programme at the University of Trieste.
To Alessio Ansuini, for our collaborations, for choosing me as his assistant for the Deep Learning course, and for giving me the possibility to realize my dream to teach in a university course.
To my colleagues at the Artificial Intelligence Student Society for the wonderful adventures that have accompanied me during the Ph.D.
To the friends and colleagues at the labs for the always enjoable company, for the lunches, for the laughs, for the after-lunch strolls.
But, above all, to Mara, for the continuous psychological support, for being close to me in the darkest moments and for always being there to listen to me.

Abstract

Pruning, in the context of Machine Learning, denotes the act of removing parameters from parametric models, such as linear models, decision trees, and ANNs. Pruning can be motivated by several necessities, first and foremost the reduction in the size and the memory footprint of a model, possibly without hurting its accuracy. The interest of the scientific community to pruning applied to ANNs has increased substantially in the last decade due to the dramatic expansion in the size of these models. This can hinder the implementation of ANNs in lower-end computers, also posing a burden to democratization of Artificial Intelligence. Recent advances in pruning techniques have empirically shown to effectively remove a large portion of parameters (even over 99%) with none to minimal loss in accuracy. Despite this, open questions on the matter still remain, especially regarding the inner dynamics of pruning concerning, e.g., the way features learned by the pruned ANNs relate to their *dense* versions, or the ability of pruned ANNs to generalize to data or environments unseen during training. In addition, pruning is often computationally-expensive and poses notable issues concerning high energy consumption and pollution. We hereby present some approaches for tackling the aforementioned issues: comparing representations/features learned by pruned ANNs, improvement in time-efficiency of pruning, application to pruning to simulated robots, with an eye on generalization. Finally, we showcase the usage of pruning for deploying, on a low-end device with limited memory, a large object detection model for face mask detection, envisioning an application of the model to videosurveillance.

Abstract in italiano

La potatura, nel contesto dell'Apprendimento Automatico, denota l'atto di rimuovere parametri da modelli parametrici come modelli lineari, alberi decisionali e Reti Neurali Artificiali (ANN). La potatura di un modello può essere motivata da numerose esigenze, primo fra tutti la riduzione in dimensione e l'occupazione di memoria, possibilmente senza inficiare l'accuratezza finale del modello. L'interesse della comunità scientifica riguardo alla potatura delle ANN è aumentato in maniera sostanziosa nell'ultimo decennio a causa dell'altrettanto cospicua crescita nella dimensione di tali modelli. Ciò può seriamente limitare l'implementazione delle ANN in computer di bassa fascia, ponendo oltretutto un ostacolo alla democratizzazione dell'Intelligenza Artificiale. Avanzamenti recenti nell'ambito della potatura hanno mostrato in maniera empirica come si può, di fatto, rimuovere una grossa porzione di parametri (a volte anche superiore al 99%) con perdita in accuratezza minima o nulla. Nonostante ciò, rimangono ancora questioni aperte in proposito, specialmente per quanto concerne le *dinamiche interne* della potatura, ad esempio riguardo alle modalità con cui le caratteristiche apprese dalle ANN potate si relazionano a quelle delle corrispondenti ANN dense, oppure all'abilità delle ANN potate di generalizzare i loro risultati a dati o ambienti non osservati durante l'addestramento. Inoltre, la potatura è spesso costosa dal punto di vista computazionale e pone notevoli problematiche connesse all'alto consumo di energia e all'inquinamento. Nel presente elaborato, esporremo alcuni approcci per affrontare i problemi sopra introdotti: comparazione di rappresentazioni/caratteristiche apprese dalle ANN potate, efficientamento temporale di tecniche di potatura, applicazione della potatura a robot simulati, con un occhio di riguardo alla generalizzazione. Infine, mostriamo un utilizzo della potatura ai fini di ridurre la dimensione di un grosso modello di riconoscimento di oggetti per il riconoscimento di mascherine facciali, implementando successivamente tale modello in un dispositivo di bassa fascia a memoria ridotta, figurando una futura applicazione del modello nel campo della videosorveglianza.

Contents

List of acronyms and abbreviations

AIMP Accelerated Iterative Magnitude Pruning

ANN Artificial Neural Network

AP Average Precision

BCE Binary Cross-Entropy

BN Batch Normalization

CC Canonical Correlation

CCA Canonical Correlation Analysis

CE Cross-Entropy

CKA Centered Kernel Alignment

CNC Centralized Neural Controller

CNN Convolutional Neural Network

DH Detection Head

DNC Distributed Neural Controller

DNN Deep Neural Network

DRQ Dynamic-Range Quantization

ES Evolutionary Strategy

FE Feature Extractorf

FIQ Full-Integer Quantization

FP False Positive

FPxx Floating-point encoding with precision xx

FPS Frames-per-Second

FN False Negative

FT Fine-tuning

GD Gradient Descent

GPU Graphics Processing Unit

IP Iterative Pruning

INTxx Integer encoding with precision xx

LR Learning Rate

LRR Learning Rate Rewind

LTH Lottery Ticket Hypothesis

ML Machine Learning

MP Magnitude Pruning

MLP Multilayer Perceptron

NBS Normalized Bures Similarity

NCS Neural Compute Stick

NLP Natural Language Processing

NE Neuroevolution

OSP One-Shot Pruning

PWCCA Projection Weighted Canonical Correlation Analysis

ResNet Residual Network

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

SMP Structured Magnitude Pruning

SVCCA Singular Vector Canonical Correlation Analysis

SVD Singular Value Decomposition

SVHN Street View House Number

TP True Positive

TN True Negative

UMP Unstructured Magnitude Pruning

UIMP Unstructured Iterative Magnitude Pruning

VSR Voxel-based Soft Robot

WR Weight Rewind

YOLO You Only Look Once

1 Introduction

In the context of Machine Learning (ML), pruning denotes a family of techniques whose function is to remove, according to given criteria, one or more parameters from parametric models. The main motivation behind pruning is the reduction in complexity of the model, which might be driven by a variety of factors [50], such as performing variable selection, increasing the generalization capability of the model, decreasing the time needed to run the model, or reducing its memory footprint, e.g., for implementation in a low-end device. Pruning is ubiquitous to the various ML techniques and has been applied succesfully to decision trees [10], linear regression [111], Artificial Neural Networks (ANNs) [70] and other family of models.

Starting from the 2010s, the ML world has witnessed an explosion in the size of ANNs, which has gone hand-in-hand with their ever-increasing accuracy, often superhuman, in popular ML benchmarks [27, 128], and with their growing popularity. The explosion has sparked the need for developing appropriate *compression* techniques able to scale down the size of the models, without hurting the final performance. Pruning, along with other techniques like quantization and knowledge distillation, is one of such approaches. In the last few years, works by Frankle and Carbin [33], Renda et al. [105], Lin et al. [75], and others yet have advanced the state-of-the-art of pruning, which is now able to produce well-performing ANNs at very high levels of sparsity (in some cases even with only 1% or less of the parameters remaining).

Despite the presence of such a large body of studies, investigations, and approaches concerning pruning, the research world still has many open questions, especially regarding the *inner working* behind the success of pruning techniques, in the same way as many theoretical properties of ANNs are still unexplained from a theoretical standpoint. Another aspect which is still researched revolves around the efficiency of pruning: in fact, producing well-performing pruned models usually

requires running procedures which can involve much more computational resources and time than the simple training of a dense ANN. Especially when referring to aspects such as energy consumption, carbon emissions, or, on another side, time-efficiency, even small advances can generate moderate-to-large improvements to these aspects.

All this considered, our research on pruning has covered two distinct macro-areas, tackling open questions and pieces of criticality insofar introduced:

> Experimental investigations whose goals were studying the inner dynamics of pruning, like the properties of features learned by pruned ANNs, and the improvement of time-efficiency of pruning.
>
> Applications of pruning techniques to specific ANN-based models with the final goal not being connected to pruning itself: pruning applied to simulated robots controlled by ANNs, and pruning an object detection model in order to deploy it in a low-end device.

In the study on the inner dynamics of pruning, presented in Section 2.4, we trained several Convolutional Neural Networks (CNNs) on computer vision tasks, then we proceeded to prune them using an iterative pruning technique based on the magnitude of the parameters. We then used four metrics (SVCCA, PWCCA, CKA, NBS) to compare the representations produced by the pruned CNNs to those produced by the dense counterparts. The aim of this investigation was to uncover possible trends or regularities in the similarities in order to gain insights on the inner working of pruning. We did not record striking results, although we noticed a generalized trend of increasing dissimilarity between the pruned and the dense versions as the pruning iteration increased.

For what concerns the study on the time-efficiency of pruning, which we showcase in Section 3.2, we operated various experiments centered on the reduction in the number of epochs required for *retraining* pruned ANNs, discovering that, often times, similar performances may be obtained by retraining much less, gaining significant speed-ups up to 4.8× with minimal decrease in accuracy.

The application of pruning to the object detection model is presented in Section 4.1. In this work, our goal was the deployment on a low-end device (a Raspberry Pi) of an object detection model for recognizing the presence of face masks on people's faces. We first trained a lightweight implementation of You Only Look Once (YOLO), a prominent, CNN-based object detector, then operated pruning with various levels of sparsity to reduce the size of the model, and finally used quantization to further speed-up the computations. Our final product was $\sim 2\times$ faster than the original one, losing only $\sim 7\%$ in precision.

The last work presented in the present manuscript is the application of pruning to simulated soft robots controlled by ANNs (Section 4.2). This analysis is motivated by the observation that, in biological neural network, pruning is a phenomenon which is fundamental for the correct functioning of the brain. We operate pruning on the ANNs at various levels of sparsity and observe the resulting performance of the robots at the task of locomotion. We conclude that pruning does not seem to have a significantly negative effect on the performances, and that does not influence the capability of the robot to adapt to different environments.

The manuscript is organized as follows:

- Chapter 2 introduces all the theoretical notions needed for the understanding of the thesis. We first give a broad overview of Supervised ML (Section 2.1) and ANNs (Section 2.2). We then introduce the *manifold assumption* (Section 2.3), a fundamental piece for perusing into the inner behaviour of ANNs, and delve into the notion of *neural representation*. Subsequently (Section 2.4), we introduce a score of metrics for the comparison of said representations. We close the chapter with an *excursus* on *model compression* (Section 2.5), addressing both pruning and quantization.

- Chapter 3 presents the two lines of experimental investigations carried out in the context of the thesis: the comparison between the representations learned by pruned CNNs (Section 3.1), and the analysis on improving the time-efficiency of pruning (Section 3.2).

- Chapter 4 covers the two lines of work concerning the application of pruning to object detection for model deployment on low-end devices (Section 4.1) and to voxel-based soft robots (Section 4.2).
- Chapter 5 finally concludes the thesis by summarizing all the lines of research and the results obtained.

2 Theoretical Framework

Mathematical Notation Throughout the current manuscript, we will use the following notation:

- *Small letters*, latin or greek, indicate *scalars* or *functions*.
- *Capital letters*, latin or greek, indicate *matrices* or *tensors*. Exceptions to this are
  - The *loss function*, introduced in Section 2.1, which will be indicated with C.
  - The total number of layers in a generic Artificial Neural Network, which we will indicate with L (Section 2.2).
  - The number of cross-correlations operated in a convolutional layer (Section 2.2.1), which we will denote with Q.
- *Calligraphic capital letters*, like $\mathcal{X}$ or $\mathcal{D}$, indicate sets. Notable sets, like the real and the natural numbers, are indicated using the capital *blackboard bold* notation (i.e., $\mathbb{R}, \mathbb{N}$).
- *Superscript indices in parentheses* indicate an element in a sequence, i.e., $a^{(i)}$ stands for the i-th element in a sequence of a's.
- *Subscripts in square brackets* stand for the *slicing* operator: generically, for a d-dimensional tensor A:
  - $A_{\underbrace{[i_1, \ldots, i_d]}_{\text{length} d}}$ indicates a generic scalar element of the tensor.
  - Indicating a sequence of d' indices in the subscript, with $d' < d$, like this $A_{\underbrace{[i_1, \ldots, i_{d'}]}_{\text{length} d' < d}}$, yields the $(d - d')$-dimensional tensor laying in the position $i_1, \ldots, i'_d$ of A.

- Subscripts with $\cdot$'s indicate that, at the corresponding dimension, all elements are to be considered: for instance, $A_{[\cdot,j]}$ yields a $(d-1)$-dimensional tensor obtained by selecting the element j in the second dimension of A, and *all* of the elements in the first, third, *etc.* dimensions.
- Slicing with colons in a given dimension indicates an interval: $A_{[i:j]}$ with $i < j$ yields a d-dimensional tensor obtained by selecting all the elements in the first dimension of A, from the i-th to the j-th included.

Notice that vectors and matrices can be though of as special cases of tensors—vectors are 1-dimensional, matrices are 2-dimensional tensors—, hence, the slicing notation can apply to them as well.

2.1 Supervised Machine Learning

The topics within this thesis mainly fall under the domain of *Supervised Machine Learning*. Machine Learning (ML) is a multidisciplinary subject, being an intersection of Computer Science, Statistics, Mathematics, and other scientific disciplines. Its aim is to produce computer programs, also called *models*, which are able to solve a class of given tasks without being explicitly programmed for that. Learning occurs from their past experience, thus improving their score on a set of performance metrics [85].

ML models learn, or are *trained*, starting from a dataset $\mathcal{D}$ which is composed of examples, on which the model must recognize specific *patterns*. In the case of *supervised learning*, the dataset is composed of samples (x, y), where $x \in \mathcal{X}$ is an *observation* over a fixed set of variables, while $y \in \mathcal{Y}$, called *target*, represent the objective of the learning; specifically, the model is tasked with learning quantitative patterns or relationships connecting $\mathcal{X}$ and $\mathcal{Y}$, such that, given a new observation x^*, it is able to predict a value $\hat{y}^*$ with a high reliability, i.e., committing a *low enough* error with respect to its observed counterpart, y^*. As data from $\mathcal{X}$ is fed to the model to formulate the predictions, its samples are then called *input* (and $\mathcal{X}$ is called *input space*). The model can be hence thought of as a function f receiving as input one

datapoint from $\mathcal{X}$ and outputting a target in $\mathcal{Y}$:

$$f : \mathcal{X} \to \mathcal{Y}. \tag{2.1}$$

In this thesis, we will be working with *parametric* models, i.e. models whose prediction is produced on the basis of *parameters* which interact, possibly in complicated fashion, with the input to produce the output. The values of the parameters are adjusted by the model itself to formulate better, more reliable predictions, to better deal with the class of task that it is required to solve. This phase of *adaptation* of the parameters is more notably known as *training*. Without loss of generality, we will indicate the set of parameters of a model as a vector $\Theta \in \mathbb{R}^p$, p being the number of parameters within the model. When needing to make explicit the parametric nature of the model, we might indicate Θ as a subscript of f, e.g., $\hat{y} = f_\Theta(x)$.

The training is usually operated on a finite sample of pairs $\mathcal{D}_{\text{train}} \doteq \{(x^{(i)}, y^{(i)})\}_{i \in \{1,\dots,n\}}$, called *training dataset*, or *trainset* in short. While other types of ML exist which do not make use of explicit targets (e.g., *unsupervised learning*), we limit ourselves to supervised learning, as this thesis operates investigations falling specifically under this domain.

For the purpose of this thesis, we will consider ML models trained to learn relationship between:

- multidimensional *continuous* observations, i.e., $\mathcal{X} \equiv \mathbb{R}^d$, where d is the dimensionality of the input space, and
- categorical targets, called *labels*, where the number of categories, or *classes*, is $k \geq 2$. In addition, we will be assuming that the classes are *indexed* from 1 to k. In this case, the task is called *classification*, in juxtaposition with *regression*, where the targets are continuous variables.

In the context of labels, we will be thinking of a generic prediction as a k-sized simplex:

$$\hat{y} \in \mathbb{R}^k, \text{ where } \hat{y}_{[j]} \geq 0 \text{ and } \sum_{j=1}^{k} \hat{y}_{[j]} = 1.$$

Hence, the prediction can be interpreted as a vector of *confidence* values, where each position expresses the confidence that the model assigns to the input being categorized to the class with the corresponding index, and this confidence may be interpreted, with a bit of a stretch, as a probability value. One possible solution for determining the predicted class might then be to pick the category corresponding to the largest element of $\hat{y}$. In this framework, we also express the ground truth label y as a k-dimensional vector: in this case, it is a *one-hot encoding* representation of the correct category. Assuming $l \in \{1, \dots, k\}$ to be the real class,

$$y_{[j]} = \begin{cases} 1 \text{ if } j = l \\ 0 \text{ otherwise.} \end{cases}$$

This representation allows for a comparison between y and $\hat{y}$ on the same dimensionality.

The objective of the training is the minimization of a real-valued term, called *loss function*, or *cost function*, and indicated with C, which is dependent upon a set of m labels $\{y^{(i)}\}_{i \in \{1,\dots,m\}}$ and their value predicted by the model $\{\hat{y}^{(i)}\}_{i \in \{1,\dots,m\}}$. The loss function is composed as an aggregation (e.g., sum, mean,...) of atomic terms $l(y, \hat{y})$, each dependent upon one label and its corresponding prediction. It increases as the dissimilarity between y and $\hat{y}$ decreases.

A common choice of l for categorical labels with $k > 2$ is the *cross-entropy* (CE):

$$\text{CE}(y, \hat{y}) = -y^\top \log(\hat{y}), \tag{2.2}$$

where, we recall, y and $\hat{y}$ are k-dimensional vectors and y is a one-hot encoding of the correct category. These atomic terms may be aggregated using a mean to compose the CE Loss:

$$\text{CELoss}(\{y^{(i)}\}_{i=1}^m, \{\hat{y}^{(i)}\}_{i=1}^m) = \frac{\sum_{i=1}^m \text{CE}(y^{(i)}, \hat{y}^{(i)})}{m}. \tag{2.3}$$

The issue of training solely based upon a performance index dependent on training data is often the source of *overfitting*, a situation in which the model performs really well, in term of loss, on samples of the

trainset, while it records underwhelming results on other points of $\mathcal{X}$. In this case, the model is also said to have a poor *generalization*, i.e., it is not able to extend its good results on unseen portions of the input space. The study of generalization makes up a vast area of research in the ML environment, a common solution is to introduce a *penalty term* on the loss function to move away the parameter from an optimal configuration on the trainset. For instance, the most common penalty term is the L2-norm regularization, also called *weight decay*:

$$\tilde{C}(\{y^{(i)}\}_{i=1}^{m}, \{f_\Theta(x^{(i)})\}_{i=1}^{m}) = \underbrace{C(\{y^{(i)}\}_{i=1}^{m}, \{\hat{y}^{(i)}\}_{i=1}^{m})}_{\text{original loss function}} + \underbrace{\lambda||\Theta||_2}_{\text{penalty}}, \quad (2.4)$$

where $||\Theta||_2$ indicates the L2 norm of Θ, and λ is a *hyperparameter*, chosen by the user and not trained by the model, to control the predominance of the penalty term over C.

2.2 Artificial Neural Networks (ANNs)

ANNs are a class of ML model developed as an abstraction of human brain, stemming from the early work of Rosenblatt [106]. Generically, ANNs are structured in *layers* of artificial *neurons* organized hierarchically from the data, forming an *input* layer, which stands at the lowest level, to the output layer. Two neurons may be connected with each other in a directed fashion: the connection—also called *synapse* in reference to the human brain—symbolizes an information flow from the first neuron to the second. Moreover, the connection is *weighted*, this meaning that the flowing information needs to be multiplied by the weight of the connection. The weights of the connections are the trainable parameters of the ANNs. Multiple incoming connections in the same neurons indicate that the incoming information is summed after this multiplication (in jargon these operations are called *multiply-add*). The simplest ANN architecture is the Multilayer Perceptron (MLP), which is characterized by *full-connectivity* between the neurons of a layer with the neurons of the previous layer in the hierarchy. Each neuron represents one of the dimensions of the layer and hence its information is characterized by a real-valued number: this means that

the input layer is composed of d neurons, each neuron holding the value for a specific dimension the data point, while the output layer is composed of k neurons in case of a k-fold classification problems, or of 1 neuron in case of regression. Between the input and the output layer, there might be additional layers with an arbitrary number of neurons. These layers are also known as *hidden layers*. Additionally, a function $\sigma : \mathbb{R} \to \mathbb{R}$ called *activation function* may be applied to the value of each neuron in a generic layer. Also, an additive parameter called *bias* may be used to offset the value of the neuron before the application of σ. Formalizing all of this, let us assume the following:

- We have a MLP with layers $0, \ldots, L$. Layer 0 is the input layer, while layer L refers to the output.
- Each layer l, $l \in \{1, \ldots, L\}$, has r_l neurons[1].
- The weights of the connections between layer $l-1$ and layer l are stored in a matrix $W^{(l)} \in \mathbb{R}^{r_l \times r_{l-1}}$.
- We assume also that each layer has an activation function σ_l which is applied element-wise to all of the neurons in the layer.
- The values of the neurons of layer l *before* the activation—also called *pre-activations*—are indicated with $z^{(l)} \in \mathbb{R}^{r_l}$.
- Similarly, the value of the neurons *after* the application of the activation function is indicated with $a^{(l)} \in \mathbb{R}^{r_l}$ and is called the *activations* of the layer.
- The bias terms are denoted with $b^{(l)} \in \mathbb{R}^{r_l}$.

Thus, we can obtain a formula for computing the pre-activations and activations of layer l:

$$z^{(l)} = W^{(l)} z^{(l-1)} + b^{(l)}, \tag{2.5}$$

[1] We specifically omit the input layer ($l \equiv 0$) as it is the only layer of a MLP with no incoming data, hence no computations are applied to obtain the value of its neurons

and

$$a^{(l)} = \sigma_l(z^{(l)}), \tag{2.6}$$

where, with an abuse of notation, the function σ_l is applied to the vector $z^{(l)}$ element-wise. We might also be viewing layer l as a function $f_l : \mathbb{R}^{r_{l-1}} \to \mathbb{R}^{r_l}$ producing a vector whose elements are obtained as indicated in Equation (2.6). Thus, the MLP can be viewed as a composition of layers, and its output a_L is the following:

$$a^{(L)} = f_L(f_{L-1}(\dots(f_1(x))\dots)). \tag{2.7}$$

Concerning *activation functions*, we must differentiate between hidden layers and output layers. The latter, in fact, need to output a data in a domain which conforms the loss function and the way the labels are represented. If we have a classification problem with $k > 2$ and we are using the CE loss, a common choice for activation function in the output layer is the *softmax*, as it outputs a standard simplex, which is the type of output we desire. For the hidden layers, instead, the user can choose between a large variety of functions, the main being Rectified Linear Unit (ReLU) or one of its variants, sigmoid, hyperbolic tangent (tanh), *etc.* In the last decade, though, ReLU has largely been favored [143] to sigmoid and tanh because of its simplicity, efficiency and the absence of a right asymptote, which was one cause of the vanishing gradient. Despite more recent variants like Leaky ReLU [135] and SiLU/Swish [32] have shown to improve ReLU across different domains, ReLU still is one of the main choices for activation functions in hidden layers of Deep Neural Network (DNN) architectures [46, 114].

2.2.1 Convolutional Neural Networks (CNNs)

Despite MLPs being very powerful ML models, they reveal mainly two notable weaknesses:

- The vectorial structure of the layers is not always able to preserve specific geometries in the data. For instance, while single-channel audio signals may be represented as vectors where each recording

is ordered by time, adding multiple channels makes a matrix representation more suitable for this type of data. Using an MLP to analyze such data may force a researcher to *flatten* the data in a single vector, thus losing the information of the multiple channels.

- The full connectivity between consecutive layers allows the MLP to be very expressive, but may result in computationally-unfeasible training phases, especially when the dimension of the input or of the hidden layers is very large. It is often possible to decrease the connectivity pattern of the model by infusing some inductive biases which are specific to the domain of the data being analyzed.

For these reasons, CNNs were introduced in order to model, specifically, images. As for the MLP, early works on CNNs by Fukushima [36] were inspired by the studies by Hubel and Wiesel [52] on the structure of the animal visual cortex. They showed that biological neural networks in the visual cortex of cats indicated that neurons were organized hierarchically such that:

- The neurons close to the retina were responding to lower-level patterns like colors or oriented edges.
- The higher-level neurons were elaborating information of lower-level neurons, progressively responding to more complicated and composite patterns.
- Also, higher-level neurons were connected only to a small amount of neurons from previous layers, and the connectivity was highly local, i.e., previous neurons were all passing information coming from a spatially close area of the visual field.

This lead ML researchers to formulate a model in which the local connectivity is formalized by means of an operation of 2D-cross-correlation (or convolution) with a kernel of fixed size and *learnable* parameters. Assuming an image I of size $h \times w$ with c channels, this is also representable by a 3D tensor of size $c \times h \times w$. We then suppose to have a

kernel K, also 3D tensor of size $c \times \kappa \times \mu$, whereas $\kappa \leq h$ and $\mu \leq w$. We call J the result of the cross-correlation, and denote with $\star$ the cross-correlation operator. The element i, j of J is obtained as

$$J_{[i,j]} = (I \star K)_{[i,j]} \doteq \sum_{c'=1}^{c} \sum_{i'=0}^{\kappa-1} \sum_{j'=0}^{\mu-1} I_{[c',i+i',j+j']} \cdot K_{[c',i'+1,j'+1]}. \quad (2.8)$$

Basically, the cross-correlation considers each possible *overlap* between the image and the kernel. For each overlap, it calculates a *weighted average* of the values of the image separately for each channel, the weights being defined by the kernel K, then sums the corresponding matrices channel-wise. The output J is hence a matrix of size $(h-\kappa+1) \times (w-\mu+1)$. An illustrated procedure of the computation contained in Equation (2.8) is depicted in Figure 2.1. The cross-correlation still reproduces the effect of multiply-add already happening in MLPs, but this is highly localized instead of being global. In fact, $J_{[i,j]}$ may be viewed as a generic neuron in a given layer whose pre-activation is determined by only a small number of neurons in the previous layer ($I_{[c',a+i',b+j']}$, with i', j' as in Equation (2.8) and $c' \in \{1, \ldots, c\}$). The size of the kernel determines the *locality* of the elements of the image influencing the generic element $J_{[i,j]}$: the larger the kernel size, the larger the area of the image affecting the value of the element. This concept has been termed *receptive field* [69], mimicking the analogous concept from neuroscience [45]. The formulation of Equation (2.8) already takes into consideration the 3-dimensional structure of images (i.e., height, width, and color channels), thus allowing for the cross-correlation to deal natively with the 3D structure of the images, without needing to flatten or reshape them, as it would have been necessary, instead, with an MLP.

A *convolutional layer* operates a fixed number Q of cross-correlations to its input. Equation (2.8) may be further augmented by adding other hyperparameters to it:

- *Padding*: adds a fixed number of rows (above and below) and columns (to the left and to the right) of the image. The added pixels may be initialized according to a variety of strategies, the common one being constant initialization, usually at 0.

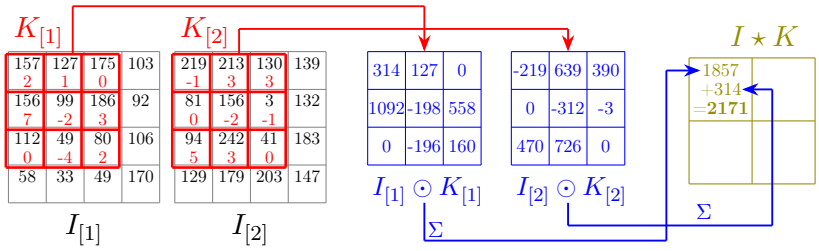


Figure 2.1: Illustrated procedure for the computation of the cross-correlation between an image I and a kernel K, indicated as $I \star K$. An image $I \in \mathbb{R}^{2\times4\times4}$ (depicted in black) and a kernel $K \in \mathbb{R}^{2\times3\times3}$ (depicted in red) are given. For computing the upper right element of the cross-correlation—i.e., $(I \star K)_{[1,1]}$—, the kernel and image are overlapped at the top-left pixels for both the matrices in the two channels. First, the Hadamard product between the two overlapping windows are computed (in blue). Then, the elements of the two matrices are summed and yield two scalars, which are then summed to produce the final result (shown in the green matrix).

- *Stride*: the possible overlaps of Equation (2.8) may be obtained by starting with the overlap between I and K at the upper-left pixel of I, then shifting the kernel one pixel to the right or down until the available overlaps are concluded. The amount of pixels by which the shifting is operated is called *stride*. The cross-correlation in Equation (2.8) has stride 1. Larger strides have the effect of skipping possible overlaps, thus reducing the size of the output J.

By considering a generic cross-correlation with a vertical stride ζ and vertical padding ρ, the vertical dimension h' of the output J is computed as:

$$h' = \left\lfloor \frac{h - k_i + 2\rho}{\zeta} \right\rfloor + 1. \tag{2.9}$$

The same holds for the horizontal counterparts. Convolutional layers can also present:

- a bias offset which can be applied to the result of the cross-correlation. As opposed to MLP layers, where each neuron has its bias, in convolutional layers each cross-correlation output has a scalar bias, which is summed to the whole output.

- an element-wise activation function to be applied after the correlation.

Recalling the notation previously used for ANNs, where each layer l had its pre-activation $z^{(l)}$ and its activation $a^{(l)}$, we can transpose this notation in the case of convolutional layers:

- The pre-activations $z^{(l)}$ and the activations $a^{(l)}$ are 3D tensors of shape $Q_l \times h_l \times w_l$, where Q_l indicates the number of cross-correlations applied by the layer l.

- The parameters $W^{(l)}$ are now the Q_l kernels. Each kernel is a 3D tensor. We can concatenate them in a 4D tensor of size $Q_l \times Q_{l-1} \times \kappa_l \times \mu_l$, where κ_l and μ_l indicate the size of kernel size.

- The bias terms $b^{(l)}$ are stored in a Q_l-sized vector.

The pre-activation for the channel $q \in \{1, \dots, Q_l\}$, i.e. the q-th position of the 3D tensor $z^{(l)}$ can be computed in two steps. First, we operate the cross correlations:

$$z_{[q]}^{(l)'} = a^{(l-1)} \star W_{[q]}^{(l)}. \tag{2.10}$$

Then, we sum the bias term for the specific channel:

$$z_{[q]}^{(l)} = z_{[q]}^{(l)'} + b_{[q]}^{(l)} \mathbf{1}_{h_l \times w_l}, \tag{2.11}$$

where $\mathbf{1}_{h_l \times w_l}$ is the ones matrix of size $h_l \times w_l$. The activation $a^{(l)}$ is, again, the output of σ_l applied element-wise to $z^{(l)}$.

CNNs make also use of *pooling layers*: layers which act by downsizing their input by dividing it in grids and applying a pre-defined scalar function to each grid; e.g., max-pooling layers apply the max function, while average-pooling layers apply the mean.

2.2.2 Training ANNs

As previously stated, the training of ML models is carried out by *tweaking* the parameters such that the resulting model $f_{\check{\Theta}}$ minimizes a loss function:

$$\check{\Theta} = \underset{\Theta \in \mathbb{R}^p}{\arg\min}\, C(\{y^{(i)}\}_{i=1}^m, \{f_\Theta(x^{(i)})\}_{i=1}^m) \tag{2.12}$$

While for some ML models, like linear regression, it is possible to find a close form solution for $\check{\Theta}$ with some specific losses, in most of the cases it is unfeasible to compute exactly the value of global minima, thus approximate iterative methods need to be employed. The most common algorithm used, especially in the context of ANNs, is Gradient Descent (GD). Starting from an initial guess Θ_0, usually determined randomly, GD works by iteratively updating the previous guess Θ_{t-1} by subtracting the opposite of the gradient of the loss attained with these parameters. In addition, a multiplier η, called *learning rate* (LR),

is applied to the gradient in order to *control* the magnitude of the step size between the current and the previous iteration:

$$\Theta_t = \Theta_{t-1} - \eta \cdot \nabla C_\Theta(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m) \tag{2.13}$$

The gradient w.r.t. the parameters Θ is usually calculated using *back-propagation* [108], assuming that the computations within the ANN and the loss function are all differentiable.

Normally, GD is applied considering $n \equiv m$, i.e., the whole training set is used to calculate the updates. This has proven though to be an ill-advised approach for ANNs as (a) the optimizer tends to get stuck in bad local minima, and (b) the usually massive number of parameters of ANNs, coupled with the often large input dimension, renders computations on modern-day computers unfeasible or anyway too time-consuming. As such, a variant of GD is more often used, called Stochastic Gradient Descent (SGD). SGD works by splitting the trainset in non-overlapping *batches* of dimension b, then using them sequentially, one in each update from Equation (2.13) (so, in SGD, $m \equiv b$). After having depleted each of the possible batches, an *epoch* is said to be concluded, and a new set of batches is obtained by re-sampling from the trainset[2]. (S)GD can be augmented by adding a physics-inspired additional term to Equation (2.13) called *momentum*, which we denote with $\Upsilon \in \mathbb{R}^p$. Momentum takes into consideration the trajectory of previous iterations for determining the direction of the current update, avoiding some *bounce-around* effects which are common with (S)GD: thus, it *smooths* the trajectories toward the optimum [97]. The update rule becomes:

$$\begin{aligned} \Upsilon_t &= \beta \cdot \Upsilon_{t-1} + \eta \cdot \nabla C_\Theta(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m) \\ \Theta_t &= \Theta_{t-1} - \Upsilon_t \end{aligned}, \tag{2.14}$$

with $\beta \in [0, 1)$ being a parameter for controlling the prevalence of the previous updates in the momentum term. $\beta = 0$ recovers the original formulation of SGD from Equation (2.13).

[2]The process of re-sampling is most often employed, but not strictly necessary for SGD, as the same set of batches may be re-used for each epoch. This procedure usually lead to sub-par results, though.

The convergence of SGD can be further improved by tweaking, mid-training, the value of the LR, thus creating a so-called *LR schedule*. This is effectively a driver to control the exploration of the parameter space, as a large LR allow for large GD steps, thus allowing for a better exploration, while lower values equate to small steps, which in term favors the *exploitation* of the current neighborhood of the parameters. A common heuristic hence is to start training with a higher value of LR (usually around 0.1) to explore the parameter space, then progressively lowering the LR at given milestones—e.g., when the loss stagnates and does not diminish. When the value of the milestones is known before training (e.g., it might be a given epoch or iteration) and the LR is reduced by a fixed multiplicative factor, this schedule is also known as *multi-step LR decay* or *annealing*.

In the context of this thesis, we present two recently developed variants of SGD which have been used in our analyses:

- Adam [61], which recovers the momentum from Equation (2.14), and introduces another heuristic, a *normalizing term* $\Xi \in \mathbb{R}^p$ whose task is to rescale the gradient's estimate in Equation (2.13) based upon a moving second moment estimate of the gradient. The update rule for Adam is the following:

$$\Upsilon_t = \frac{\beta_1 \Upsilon_{t-1} + (1-\beta_1)\nabla_\Theta C(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m)}{1-\beta_1^t} \tag{2.15}$$

$$\Xi_t = \frac{\beta_2 \Xi_{t-1} + (1-\beta_2)\nabla_\Theta C(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m)^2}{1-\beta_2^t} \tag{2.16}$$

$$\Theta_t = \Theta_{t-1} - \eta \cdot \Upsilon_t \oslash (\sqrt{\Xi_t} + \mathbf{1}_p \epsilon), \tag{2.17}$$

  where β_1 is β from Equation (2.14), and $\beta_2 \in (0,1)$ is a hyperparameter for controlling the prominence of old values in the moving average in Equation (2.16). The squared gradient here is referring to the element-by-element squaring of such vector, ϵ is a small constant added for numerical stability, and $\oslash$ indicates the vector element-by-element *division*, i.e., assuming p-sized vectors

A and B, $A \oslash B = [a_{[1]}/b_{[1]}, \dots, a_{[p]}/b_{[p]}]^\top$. Adam was developed to tackle some shortcomings of SGD, namely (a) the need of a LR schedule for controlling exploration/exploitation: Adam effectively incorporates a variable LR by means of the two terms Υ_t—Equation (2.15)—and Ξ_t—Equation (2.16)—, and (b) the fact that the GD step direction is dominated by parameters having large gradients, which, in ANNs, are usually those belonging to the last layers, which means that earlier layers experience very small updates: the normalizing term Ξ_t tries to enforce exploration also in other directions by penalizing parameters with large gradient variance.

- RAdam [77] augments Adam by adding an additional variance rectification term to Equation (2.16). Its development was prompted by observations that Adam suffers from large variance in the initial stages, which may require an initial *warmup* phase during training, consisting in a number of iterations in which the LR starts from a very low value and is rapidly increased to the initial value of the schedule. RAdam is formulated as such:

$$\rho_\infty = \frac{2}{1-\beta_2} - 1 \tag{2.18}$$

$$\upsilon_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}} \tag{2.19}$$

$$\Upsilon_t = \frac{\beta_1 \Upsilon_{t-1} + (1-\beta_1)\nabla_\Theta C(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m)}{1-\beta_1^t}$$

$$\Xi_t = \frac{\Xi_{t-1}}{\beta_2} + (1-\beta_2)\nabla_\Theta C(\{y^{(i)}\}_{i=1}^m, \{f_{\Theta_{t-1}}(x^{(i)})\}_{i=1}^m)^2$$

$$\Xi'_t = \sqrt{(1-\beta_2^t)\mathbf{1}_p \oslash \Xi_t} \tag{2.20}$$

$$\Theta_t = \begin{cases} \Theta_{t-1} - \eta \upsilon_t \Upsilon_t \odot \Xi'_t, & \text{if } \rho_t > 4 \\ \Theta_{t-1} - \eta \Upsilon_t & \text{otherwise} \end{cases} \tag{2.21}$$

where, in Equation (2.20), the square root is to be applied elementwise, and $\mathbf{1}_p$ indicates the ones-vector of size p, while the symbol

$\odot$ in Equation (2.21) stands for the Hadamard product. The rectification to the variance normalizing term introduced in Equation (2.20) heuristically reproduces the warmup phase, which does not need to be ran explicitly during training. In addition, the condition in Equation (2.21) allows to avoid degenerate cases in which the variance of Ξ'_t is intractable.

Despite the fact that Adam and RAdam, due to their adaptive nature, should avoid the need for LR schedulers, [105] report advantages using a multi-step decay with Adam; morover, we [147] found it beneficial in speeding-up the training for RAdam as well.

Neuroevolution (NE): an alternative approach to learning for ANNs

NE is an approach for learning both structures and parameters of ANNs which relies on principles of Evolutionary Algorithms (EA) instead of statistical learning for obtaining well-performing models. The main advantage of NE over training with backpropagation is generality, as NE allows for learning without a proper target (i.e., a loss function in the case of statistical learning), or with loss functions which are not differentiable. Moreover, NE can also support the morphological evolution of the ANN, i.e., synapses or neurons can be newly created or removed, like in the NEAT algorithm [100, 120]. NE does not train ANNs, rather it *evolves* them, providing a genetic approach to producing these models. This approach supports *population-based* techniques, that evolve a *population* of individuals—which, in NE, are ANNs—by applying *natural selection* to the individual in the population: the individuals are evaluated on a metric called *fitness*, which measures their performance at completing one or more tasks of interest. After a fixed time interval, the best individuals may be then *selected*, removing the others, and a new population can be produced, e.g., by mutating the surviving individual, or by breeding them. This scheme of fitness evaluation, selection and production of an offspring is iterated for an arbitrary amount of times. The new population is also called a *generation*, so, simplifying, we can say that the algorithms runs for a number of generations.

Of the many existing NE algorithms, we highlight Evolution Strategy (ES) [101], which is a population-based generic technique for evolution-based optimization of the fitness, this meaning that we have a population of m ANNs which are mutated (or bred) to reach the optimum. An ANN is represented as a vector of connections $\Theta^{(i)} \in \mathbb{R}^p$. Mutations can be performed by randomly tweaking the parameters, e.g. by adding Gaussian noise to them. Notice that this rudimentary approach does not support morphological evolution, as the ANN always has p synapses, and the mutation scheme does not allow for modifying the neurons that each synapse connects. For each generation, we can select a number of top-performing ANNs, which we can mutate to produce the offspring for the next generation. If we wish for our population not to decrease in number, each ANN can undergo multiple mutations, each time producing a different ANN as a result.

Despite its flexibility and the possibility of adopting a population-based learning, NE is though currently underutilized in the DL community due to a difficulty in producing competitive ANNs: especially when the models are deep, in fact, DNNs trained with SGD tend to reach better solutions in terms of accuracy [37]. Nevertheless, we have used NE when applying pruning to simulated robots (see section 4.2), in a context in which the problem setting did not consider a differentiable fitness function, and, thus, SGD could not be used directly.

2.2.3 Additional ANNs modules

ANN architectures can be further expanded with the addition of modules that can help to achieve faster convergence or better accuracy.

Batch Normalization (BN)

BN [53] is an ANN layer which acts, in the context of SGD training, by transforming the distribution of the input of a layer to a normal distribution with learnable mean and variance. The normalization acts in two phases:

1. The input batch $A \in \mathbb{R}^{b \times r}$, r being the generic number of neurons of the previous layer, is normalized to have mean 0 and variance

1 regardless of its initial distribution: $A_{\text{norm}} = (A - \text{mean}(A)) \cdot \text{var}(A)^{-1/2}$.

2. The normalized input is shifted and rescaled according to two learnable parameters γ_{bn} and β_{bn}: $A_{\text{bn}} = \gamma_{\text{bn}} A_{\text{norm}} + \beta_{\text{bn}}$.

Despite an experimentally provable benefit to convergence, the theoretical reasons for the effectiveness of BN are still debated in the literature [8, 53, 112]. BN modules are often employed before an MLP layer to normalize its input; conversely, in CNNs, it has been found beneficial [53] to use them in convolutional layers *after* the cross-correlation operations, but *before* applying the activation function.

2.2.4 Common CNN architectures

Within the context of this thesis, we will be using mainly three CNN architectures: (a) VGGs [114] and (b) ResNets [46], which are CNNs for image classification, and (c) YOLOv4 [9, 103], an ANN for object recognition. The task of *image classification* is to classify an image in a single category according to its content, while *object recognition* in the context of computer vision refers to the classification of instance(s) of possibly multiple objects throughout an image, while simultaneously localizing these instances within the image.

VGG

VGG or VGGNet is a class of CNNs for image classification first implemented by Simonyan and Zisserman [114]. The core concept behind VGG is the usage of a cascade of *convolutional blocks*, i.e., blocks of layers composed of 2 to 4 convolutional layers followed by a max-pooling for downscaling. The convolutional layers have usually a kernel size of 3 and padding of 1, effectively leaving the spatial dimension (i.e., height and width) of the incoming data untouched. Additionally, the convolutional layers within the same block all share the same number of cross-correlations Q, thus producing outputs of the same shape. Q is usually a power of 2, starts at 64, is doubled after each block, and

is capped at 512. Since it is shown [142] that CNNs tend to reproduce the compositionality of human vision (i.e., neurons close to the retina learn low-level features, while later neurons combine these features in higher-level ones, as explained also in Section 2.2.1), the effect of the increasing number of channels of the output is that a VGG-CNN learns few lower-level features like edges, but has an extensive degree of freedom to combine these features into a large number of higher-level ones, which allows the CNN to achieve what then were state-of-the-art results on the large-scale image classification benchmark dataset ImageNet [27, 109]. After the cascade of convolutional blocks, the output is flattened in order to be fed to a cascade of MLP layers. While the original implementation of VGG was considering the usage of 3 of such layers—2 hidden and one output layer—, later investigations led by Springenberg et al. [117] advocated for dropping hidden MLP layers in CNNs, so the versions used nowadays include only the output layer after the flattening. Also, the flattening was replaced by a channel-wise averaging of the last convolutional layer's output, sometimes called Global Average Pooling—GAP, thus allowing for images of arbitrary size to be evaluated by the CNN, instead of imposing one fixed size for all images. Eventually, BN modules were later introduced to the VGG architectures to aid their convergence. VGGs are named by prepending this acronym to the number of convolutional or MLP layers in the model; thus, "`VGG16`" refers to a VGG architecture with 16 layers. With the aforementioned removal of the 2 MLP layers, the `VGG16` has actually 14 layers, but the nomenclature has not been updated accordingly. Simonyan and Zisserman [114] introduced four VGG architectures: `VGG11`, `VGG13`, `VGG16`, and `VGG19`. Deeper architectures did not improve the results due to the *vanishing gradient* issue, occurring in too-deep ANNs: the gradient, which is being backpropagated, slowly vanishes towards 0 as it flows back to the earlier layer. This results in the loss not decreasing as the parameters in the initial layers cannot be trained due to the gradient being 0—see Equation (2.13). A schematic depiction of `VGG16` is shown in Figure 2.2. It is composed of 5 convolutional blocks, having respectively 2, 2, 3, 3, and 3 convolutional layers each.

In our investigations, we mainly experimented with the architectures

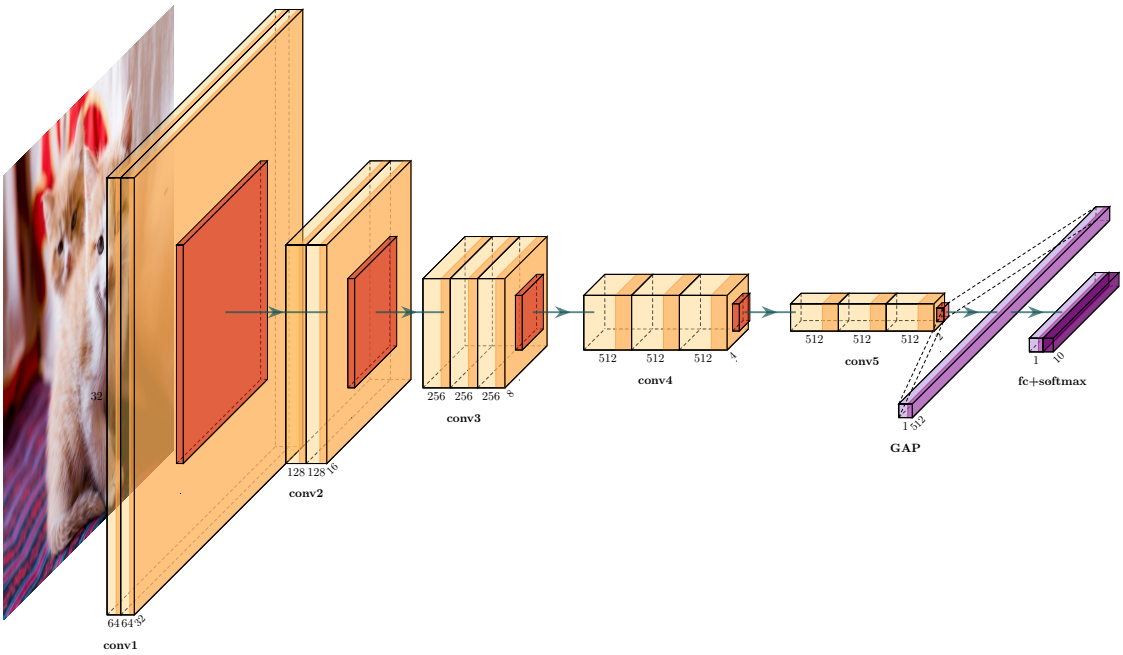


Figure 2.2: Schematic representation of the "modern" VGG16 architecture, with 13 convolutional layers. Convolutional layers are depicted in yellow, while max pooling is depicted in red, GAP and MLP layers are drawn in teal, the final softmax in gray. Activation functions are ReLU everywhere except for the final MLP layer; all convolutional layers include a BN step between the cross-correlations and the activation function. Numbers at the bottom of layers indicate the number of channels of the output of the same layer, while the number below the diagonal side indicate the spatial dimension (assuming $h = w$), supposing an initial image of size $3 \times 32 \times 32$.

`VGG16` and `VGG19`.

ResNets

Residual Networks, or, in short, *ResNets*, are a class of ANNs introduced by He et al. [46], initially designed for image classification. ResNets are characterized by multiple blocks implementing *skip connections*. These are mechanisms where the input to one layer gets *duplicated* and flows along two different paths: the first one passes through one or more layers, the second one—which is the proper skip connection—leaves its data untouched and joins the first one via concatenation or sum. Whenever the shape of the two data are not conformable, minimal operations on the data of the skip connection are allowed in order to render said concatenation or sum possible. The advantage of the skip connection lies in the gradient flow, as they keep the gradients "alive" avoiding the issue of the vanishing gradient, thus allowing for the construction of very deep ANNs, much deeper than, e.g., VGGs. Despite the concept of skip connection being ubiquitous in the context of DL, ResNets were originally designed to be CNNs. He et al. [46] designed two types of *Residual Blocks* implementing skip connections: one called *BasicBlock*, the other *Bottleneck*. Referencing Figure 2.3, BasicBlocks contain two convolutional layers having the same hyperparameters (i.e., kernel size, padding, *etc.*), while Bottlenecks contain three layers which contract and then expand the channel dimension of the data passing through. Bottlenecks were proven to be more efficient than BasicBlocks when dealing with very DNNs. As for VGGs, ResNets are named appending the number of convolutional layers to the "`ResNet`" acronym. The authors introduced five ResNet architectures: `ResNet18` and `ResNet34` stack BasicBlock, while `ResNet50`, `ResNet101`, and `ResNet152` use a large number of Bottleneck blocks. The architecture of the ResNet18 is schematized in Figure 2.4.

YOLOv4 for object detection

Object detection is defined as the task of recognizing within an image instances of a pre-defined set of categories, while performing a coarse

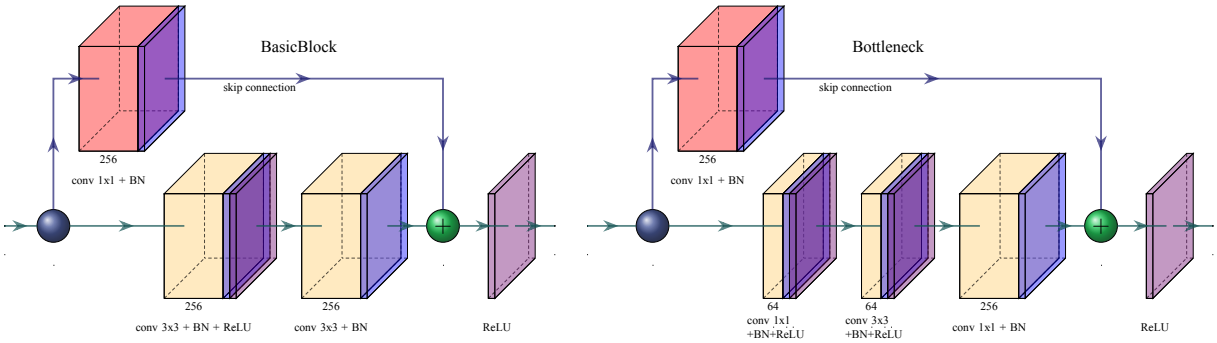


Figure 2.3: Depiction of ResNet's BasicBlock and Bottleneck with a base width of 256 (i.e., they output data with 256 channels). Convolutional layers—"conv"—are indicated in yellow or red. "conv n×n" indicates that the layer uses kernels of size $n \times n$. The convolutional layers in yellow have always a stride of 1 and a padding of 1 in case the kernel has size 3×3, 0 otherwise. The red convolutional layer in the skip connection has a stride of 2 and a padding of 0. It is present only when the block operates a *downsampling* of the spatial dimensions. In this case, also the first convolutional layer in yellow has a stride of 2 instead of 1. BN—in blue in the figure—is applied after each convolutional layer.

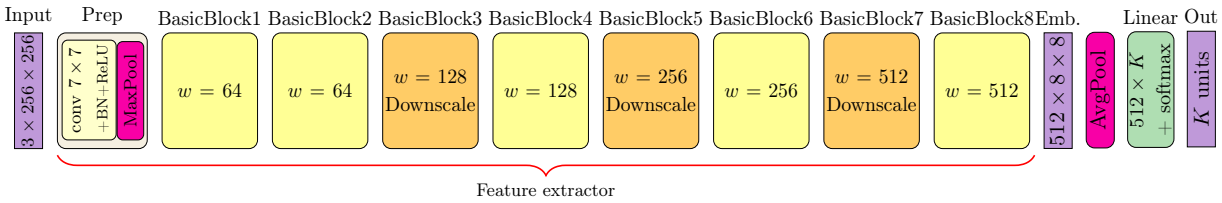


Figure 2.4: Schematic representation of the ResNet18 architecture [46]. The incoming data ("Input"), in our case of size $3\times256\times256$, passes through a *preparatory block* ("prep") which has a 7×7 convolutional layer with stride 2, padding 3, a subsequent BN phase and a ReLU activation function. Following that, the data is shrunk via a max pooling layer ("MaxPool"). Then, it flows through a cascade of BasicBlocks, progressively shrinking in spatial dimension and increasing in channels. "w" indicates the *base width* of the block, while "Downscale" specifies that the block is responsible for downsampling the spatial dimension of the data. Then, the blocks produce an embedding ("Emb.") of size $512 \times 8 \times 8$. This is averaged along the spatial dimensions ("AvgPool"—*Average Pooling*) outputting a vector of 512 elements. This, in turn, is processed by a linear layer with softmax activation. It projects the vector into a K-dimensional space, where K is the number of classes. The softmax turns it into a simplex, which is the final output ("Out") of the model.

localization, e.g., drawing bounding boxes loosely containing said instances. The literature [121] distinguishes between (a) *one-shot* detectors, i.e., models which perform localization and recognition at the same time, and (b) *two-shot* detectors, i.e., models which first identify the area and then, in a second moment, operate the classification of the content within this area. Models falling within (a) usually focus on accuracy at the expense of inference speed; on the other hand (b) tend to be faster, but at the cost of inaccurate classification or position of bounding boxes.

Within the context of this thesis, we concentrate on You Only Look Once (YOLO), a family of CNNs for one-shot object detection first introduced by [103]. Numerous variants of YOLO have been developed after its first implementation, either as *new releases*—like YOLOv3 [102] and YOLOv4 [9]—or as improvements for specialized tasks, like [42, 110], or for increased temporal efficiency [2, 59]. The common trait connecting all these models is the presence of a *feature extractor* (FE) or *backbone*, usually a CNN, and some *detection heads* (DHs) tasked with formulating the prediction. FEs, similarly to the other CNNs previously introduced, are a cascade of convolutional layers, even if recent specimens, like the one introduced by Jiang et al. [59], employ additional blocks such as *attention modules* [133]. The task of FEs is, similarly to CNNs, the elicitation of feature useful for the task at hand. DHs, instead, combine the features to formulate the predictions. YOLO-based architectures usually employ more than one DH, and each DH is tasked with formulating a predefined number of proposal predictions. Each proposal is a fixed vector containing:

- The four coordinates of the bounding box—$x, y, \Delta x, \Delta y$: x, y are the horizontal and vertical coordinates, $\Delta x, \Delta y$ are the horizontal and the vertical offsets from x, y.
- A simplex of k elements, one per category (as in Section 2.1 in the case of classification), used for formulating the prediction of the category contained within the box.
- A scalar $o \in [0, 1]$ specifying the *confidence* in the prediction: when the confidence falls below a certain threshold (as defined

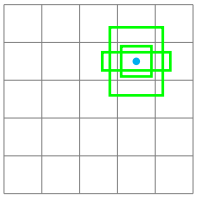


Figure 2.5: Formulation of prediction in a DH of YOLOv4 depending on the anchor boxes. In the image, the spatial dimensions of the output are 5×5, and the number of anchor boxes is 3. The cell at position $(2, 4)$—whose center is indicated with the cyan dot—outputs three different prediction proposals. The bounding boxes, drawn in green, have a fixed size, which is the same for all the other cells in the grid. In order to match the predictions with the ground truth, the whole matrix is then scaled up to match the spatial dimension of the input, upscaling the size of the bounding boxes as well.

by the user), the prediction proposal is ignored.

Usually, YOLO-based models are built in order to formulate grids of predictions proposals at multiple scales, the stratagem for forming these proposals being specific for each architecture.

For instance, YOLOv4 has three DHs. Each DH formulates a prediction at different scales: for instance, in the original implementations, the spatial dimension of the output of the DHs are $52 \times 52, 26 \times 26$, and 13×13. We will call each element of these *grids* a *cell*. Each *cell* at these dimensions outputs n_{anchor} different proposals: the bounding box size for each proposal is fixed and depends upon a hyperparameter called *anchor boxes*, which basically biases the model to always output bounding boxes at a pre-defined size, scaled according to the resolution of the DH. In YOLOv4, $n_{\text{anchor}} = 3$. A depiction of this process is visible in Figure 2.5. These predictions proposals are then scaled up to equal the spatial dimension of the input, matched to the closest ground truth(s), and *non-maximal suppression* is applied to avoid duplicate predictions.

2.3 How does a ML model learn? The manifold assumption

A common assumption concerning geometrical properties of the data space $\mathcal{X}$ is that the data is *embedded* in a topologically-complex lower-dimensional manifold [81]. Learning the relationships between the covariates and the labels in $\mathcal{X}$ is a complicated task, while unravelling this lower-dimensional manifold would ease the task. While some models—e.g., linear regression—try to learn these relationships directly in the input space, there exist other models which, instead, try to compute this process of unravelling before operating the final classification or regression. Maybe the most famous family of these models, and also the one which best exemplifies this process, are the Support Vector Machines [21] for solving classification problems. The way these models operate can be roughly thought of as applying a transformation thanks to a kernel function—chosen as a hyperparameter—then applying linear separation to the data embedded in this new space. ANNs also incorporate this process, but with a few differences:

- The unravelling of this complex manifold is not operated one-shot, but is achieved progressively layer-by-layer [3].
- The transformations operated by each single layer are not defined by the user as a hyperparameter, but are effectively learned by the ANN, being characterized by the set of parameters belonging to the layers.

Each layer can hence be seen as one gear of an engine whose final objective is to map the data from the input space $\mathcal{X}$ to a (possibly) less-complicated embedding where the mapped data are linearly separable in case of classification, or linearly *fittable* in case of regression. Thus, gaining insights on features of the representations produced by hidden and output layers can lead to advances in the understanding of the inner working principles of ANNs.

2.3.1 Computing latent representations

One possible line of research going in the way of gaining insights on the inner workings of ML models is related to the comparison between *latent* manifolds produced by the models. Before delving into the techniques for operating these comparisons, though, it is first necessary to explain how to produce the representations. For the scope of this thesis, we will limit ourselves to the case of ANNs: in this case, we will also call these representation *neural representations*. Let us consider the simple case of a MLP. Recalling the notation from Section 2.2, a generic layer l is composed of r_l neurons and produces an activation $a(l) \in \mathbb{R}^{r_l}$. A representation of this layer may hence be produced by considering all possible infinite combinations of inputs from $\mathcal{X}$ and observing the response of all the r_l neurons to these inputs. This is though infeasible as (a) the black box nature of ANNs makes extremely difficult the computation of functional representations of neurons, and (b) it is impossible to get an infinite, exhaustive sample from $\mathcal{X}$. Thus, the solution is to use a finite, possibly representative sample of inputs $\{x_i \sim \mathcal{X}\}_{i \in \{1,\dots,m\}}$ and use the response of the layer to each of these inputs to construct a representation of the layer. Considering that, for each data point, a vector of activations $a^{(l)} \in \mathbb{R}^{r_l}$ is produced, we can pack the m output vectors into a matrix $A^{(l)} \in \mathbb{R}^{m \times r_l}$. Thus, getting the latent representation equates to computing the matrix of activations produced by the layer as output of the representative sample $\{x^{(i)}\}_{i \in \{1,\dots,m\}}$.

Representations of CNNs

Convolutional layers differ from MLP layers in the dimension of the output: while the latter produce vectors in response to a single input, the former produce a 3D tensor. Thus, considering the same sample $\{x^{(i)}\}_{i \in \{1,\dots,m\}}$, a representation of a convolutional layer can be thought of as a 4D tensor $A^{(l)} \in \mathbb{R}^{m \times Q_l \times h_l \times w_l}$. This tensor may be reshaped in a matrix by *combining* more dimensions into one, as we will see in the next sections.

2.4 Techniques for comparing neural representations

The comparison of *neural representations* is a hard task as (a) the metrics used need to abide by a number of desirable properties, which commonly used matricial metrics, like the cosine similarity, do not have, and (b) the metrics need to extend to cases where the neural manifolds are produced by ANNs with different numbers of neurons. Comparing the representation can be seen as a way to study the *features* learned by ANNs: after all, as we have seen in Section 2.3.1, the representations are *expressions* of the neurons in response to a finite dataset; neurons *are* the features of ANNs, and studying their meaning, analyzing them semantically can reveal important information on the inner dynamics of these models [40, 93].

Let $A^{(1)} \in \mathbb{R}^{m \times r^{(1)}}$ and $A^{(2)} \in \mathbb{R}^{m \times r^{(2)}}$ be two representations of generic MLP layers having $r^{(1)}$ and $r^{(2)}$ neurons respectively, with $r^{(1)} \gtrless r^{(2)}$. We will first restrict to the cases of MLPs, then extend the techniques for handling convolutional layers as well. Let us introduce a similarity metric M that allows us to compare these two generic representations. We suppose that this metric is *relative*, i.e., it outputs a value which is upper- and lower-bounded in the sense that the upper value indicates maximum similarity, while the lower value stands for maximum dissimilarity. Kornblith et al. [63] summarize the desirable properties that M must abide to in order to be considered a metric for comparing neural representations.

Rotational invariance Given a trained ANNs, it is possible to obtain a functionally identical ANN by carefully permuting the parameters layer-by-layer, and also allowing sign flips at parameter level [16]. Thus, we wish M to incorporate rotational invariance with respect to the neurons of the layers to be compared: given signed permutation matrices $R^{(1)} \in \mathbb{R}^{r^{(1)} \times r^{(1)}}, R^{(2)} \in \mathbb{R}^{r^{(2)} \times r^{(2)}}$ that changes the order of the neurons in a layer—i.e., it changes the order of the columns in the representation matrix, allowing also sign flips, we wish

$$M(A^{(1)}, A^{(2)}) = M(A^{(1)}R^{(1)}, A^{(2)}R^{(2)}). \tag{2.22}$$

Invariance to isotropic scaling The metric M should, moreover, be invariant to isotropic scalings: given $\alpha_1, \alpha_2 \in \mathbb{R}^+$, we expect

$$M(A^{(1)}, A^{(2)}) = M(\alpha_1 A^{(1)}, \alpha_2 A^{(2)}).$$

Intuitively, this requisite makes sense because scaling all of the parameters of an ANN does not modify its final output. That equates to requiring that the similarities are not relying upon the magnitude of the activations, but mainly on their pair-wise orientations.

Non-invariance to generic invertible linear transformation Despite the rotational invariance being a desirable property, Kornblith et al. [63] make the case for not having metrics invariant to generic invertible linear transformations. These are representable by full-rank matrices $S_1 \in \mathbb{R}^{r_1 \times r_1}, S_2 \in \mathbb{R}^{r_2 \times r_2}$. We generally wish for

$$M(A^{(1)}, A^{(2)}) \neq M(A^{(1)} S_1, A^{(2)} S_2). \tag{2.23}$$

The practical reason for this requirement is that, as demonstrated in [63], given two representations A, B having row size m and rank m, another generic representation C of row size m, and a metric M invertible to linear transforms, $M(A, C) = M(B, C)$. This equates to saying that the metric will fail to work as expected with representations having a number of neurons larger than the number of data points. This situation is common especially with CNNs, as the number of total neurons can be very high: for instance, in a ResNet18, depending upon the size of the input, the output of the first convolutional layer can easily have more than 1 000 000 neurons, which would require to have more than 1 million data points to produce a representation in the context of a metric invariant to invertible linear transformations.

2.4.1 Canonical Correlation Analysis (CCA)

Canonical Correlation Analysis [51] is a multivariate statistical tool to compare two generic representations of *phenomena* both formed from the same set of m statistical units. This is indeed the case of two

neural representations, hence CCA can be used to investigate relationships between these representations as well. Given representations $A^{(1)} \in \mathbb{R}^{m \times r^{(1)}}$ and $A^{(2)} \in \mathbb{R}^{m \times r^{(2)}}$, let $\tilde{r}$ be $\min(r^{(1)}, r^{(2)})$. We *center* representations w.r.t. their column mean, i.e., $A^{(1)} \leftarrow A^{(1)} - \mathbf{1}_m^\top A^{(1)}/m$.

The goal of CCA is to find two linear transforms $T^{(1)} \in \mathbb{R}^{r^{(1)} \times \tilde{r}}$ and $T^{(2)} \in \mathbb{R}^{r^{(2)} \times \tilde{r}}$ projecting the two representations onto a common two-dimensional space of shape $m \times \tilde{r}$ where the two representation are the more *aligned as possible*:

$$\begin{aligned} \Phi^{(1)} &= A^{(1)} T^{(1)} \\ \Phi^{(2)} &= A^{(2)} T^{(2)}. \end{aligned} \tag{2.24}$$

We denote with $\Phi^{(k)}_{[\cdot,i]}$ the i-th column of $\Phi^{(k)}$, with $k \in \{1,2\}$ and $i \in \{1,\dots,\tilde{r}\}$. This vector is also called *canonical vector*. As a requisite for the *alignment*, we want the canonical vectors within the same projection be pair-wise perpendicular and have each unitary norm:

$$\begin{aligned} \Phi^{(k)}_{[\cdot,i]} &\perp \Phi^{(k)}_{[\cdot,j]} \\ ||\Phi^{(k)}_{[\cdot,i]}||_2 &= 1, \end{aligned}$$

with $k \in \{1,2\}$ and $i,j \in \{1,\dots,\tilde{r}\},\ i \neq j$. In addition, let us also define the *canonical correlation* (CC), denoted with ρ, as the pair-wise Pearson correlation between the two canonical vectors with identical index

$$\begin{aligned} \rho_i \doteq \mathrm{CORR}(\Phi^{(1)}_{[\cdot,i]}, \Phi^{(2)}_{[\cdot,i]}) &= \frac{{\Phi^{(1)}_{[\cdot,i]}}^\top \Phi^{(2)}_{[\cdot,i]}}{||\Phi^{(1)}_{[\cdot,i]}||_2 \cdot ||\Phi^{(2)}_{[\cdot,i]}||_2} \\ &\overset{\text{orth.}}{=} {\Phi^{(1)}_{[\cdot,i]}}^\top \Phi^{(2)}_{[\cdot,i]} \end{aligned} \tag{2.25}$$

with $i \in \{1,\dots,\tilde{r}\}$. Starting with $i = 1$, we wish to find $T^{(1)}, T^{(2)}$ to maximize ρ_1. The following coefficient $\rho_2, \dots, \rho_{\tilde{r}}$ will be each constructed maximizing the *residual* Pearson correlation. Thus, $\Phi^{(1)}$ and $\Phi^{(2)}$ will be two orthogonal matrices with columns exhibiting a decreasing pairwise correlation: $\rho_1 \geq \rho_2 \geq \cdots \geq \rho_{\tilde{r}}$.

The matrices $T^{(1)}, T^{(2)}$ may be found in one-shot by applying the Singular Value Decomposition (SVD) to the matrix $\Sigma_{11}^{-1/2}\Sigma_{12}\Sigma_{22}^{-1/2}$, where Σ_{11} and Σ_{12} are the covariance matrix of $A^{(1)}$ and $A^{(2)}$ respectively, and Σ_{12} is the cross-covariance between $A^{(1)}$ and $A^{(2)}$:

$$\Sigma_{11}^{-1/2}\Sigma_{12}\Sigma_{22}^{-1/2} = UDV^\top. \tag{2.26}$$

It can be shown that the CCs are stored as entries in the diagonal matrix D: $\rho_i = D_{[i,i]}, i \in \{1, \dots, \tilde{r}\}$.

Since the CCs measure the correlations between canonical vectors, they can be averaged to form a metric of "alignment" between the two representations. Such index is called *Mean CCA Similarity* [98]:

$$\text{MeanCCASimilarity} \doteq \frac{\sum_{i=1}^{\tilde{r}} \rho^{(i)}}{\tilde{r}}. \tag{2.27}$$

Kornblith et al. [63] proved Mean CCA Similarity to be invariant to generic invertible linear transformations, thus rendering this metric unsuitable for comparing neural representations. In the case of convolutional layers, in order to apply Mean CCA Similarity, we would need to reshape the 4D representations into a matrix. A logical reshaping is to collapse the spatial dimensions h, w onto the channel dimension Q:

$$(m \times Q \times h \times w) \longrightarrow (m \times hwQ). \tag{2.28}$$

This reshaping, though, might easily produce a matrix where the number of *neurons* is larger than the number of data points m, thus falling into the criticality previously presented in Section 2.4. For this reason, Raghu et al. [98], propose to reshape the representations of convolutional layers to avoid this situation. Their solution is to process this reshape, incorporating the spatial dimensions into the dimension of data points:

$$(m \times Q \times h \times w) \longrightarrow (mhw \times Q). \tag{2.29}$$

This operation is motivated by the fact that, in CNNs, due to the *weight sharing* induced by the sliding kernel in the cross-correlation operation, each pixel of the $h \times w$ map for a given channel q is produced by the

same set of parameters (i.e., the kernel). Due to this, a *neuron* or *feature* of the CNN is the whole $h \times w$ channel, and not one pixel of it. This is the case, for instance, in the subject of *feature visualization* [93]. We give a further motivation and contextualization in favor of this form of resizing in our experimental investigations—see Section 3.1.9.

Singular Vector CCA (SVCCA)

Raghu et al. [98] observed experimentally that most of the neurons contribute little to the representations of the corresponding layers: thus, they proposed to recover the subspace of *relevant neurons* by first decomposing both $A^{(1)}$ and $A^{(2)}$ using SVD, then reducing the dimensionality by choosing an appropriate number of singular values, e.g., those explaining 99% of the variance. Formally, we decompose $A^{(1)}$ and $A^{(2)}$ using SVD:

$$A^{(1)} \overset{\text{SVD}}{=} \Pi^{(1)}\Delta^{(1)}\Gamma^{(1)}$$
$$A^{(2)} \overset{\text{SVD}}{=} \Pi^{(2)}\Delta^{(2)}\Gamma^{(2)}.$$

We find the threshold index of singular values explaining a fraction $\tilde{\sigma}$ of variance:

$$r_1^* = \min_{r \in \{1,\dots,r^{(1)}\}} \left[\sum_{i=1}^{r} (\Delta^{(1)}_{[i,i]})^2 \geq \tilde{\sigma} \right]$$
$$r_2^* = \min_{r \in \{1,\dots,r^{(2)}\}} \left[\sum_{i=1}^{r} (\Delta^{(2)}_{[i,i]})^2 \geq \tilde{\sigma} \right]$$

We reduce the dimensionality of the representations according to r_1^* and r_2^*:

$$\tilde{A}^{(1)} = \Pi_{[\cdot,1:r_1^*]}\Delta_{[1:r_1^*,1:r_1^*]}$$
$$\tilde{A}^{(2)} = \Pi_{[\cdot,1:r_2^*]}\Delta_{[1:r_2^*,1:r_2^*]}.$$

Now, CCA can be computed on these two matrices thus obtaining a new set of CCs $\{\rho_1, \dots, \rho_{\min(r_1^*, r_2^*)}\}$. This whole procedure is called

SVCCA, and the metric obtained from averaging the CCs is named *Mean SVCCA Similarity*.

Raghu et al. [98] state that SVCCA is advantageous w.r.t. CCA by highlighting an example. Supposing that two representations $A^{(1)}$ and $A^{(2)}$ are one the subspace of the other (we suppose $r^{(1)} < r^{(2)}$), the Mean CCA Similarity would be equal to 1, as the matrices are perfectly aligned on $\tilde{r} = \min(r^{(1)}, r^{(2)})$ dimensions. This is not ideal, though, as this completely disregards the information coming from the other $\tilde{r} - r^{(2)}$ dimensions of $A^{(2)}$. By incorporating the initial SVD, though, SVCCA can give a different view of the similarity between the two representations. In addition, Kornblith et al. [63] show how, while CCA is invariant to invertible linear transformations, SVCCA generically drops that invariance, thus rendering SVCCA superior to CCA for the sake of comparing neural representations.

Projection Weighted CCA (PWCCA)

PWCCA [86] was designed as an improvement to the Mean (SV)CCA Similarity, which, as of Equation (2.27), weights equally all CCs. The core idea is to re-weight each CC according to the contribution of the corresponding canonical vector in the original representation of the layer. Assuming that the representation $A^{(1)}$ is the one with the smaller number of neurons, i.e., $\tilde{r} = r^{(1)}$, a set of weights α are defined as

$$\alpha^{(i)} \doteq \sum_{j=1}^{\tilde{r}} \left| {A^{(1)}_{[\cdot,j]}}^{\top} \Phi^{(1)}_{[\cdot,i]} \right|,$$

with $i \in \{1, \ldots, \tilde{r}\}$. If $\tilde{r} = r^{(2)}$ instead, columns from $A^{(2)}$ and $\Phi^{(2)}$ must be used. The coefficient α_i measures the (unsigned) alignment between the i-th canonical vector and all of the neurons of the representation $A^{(1)}$, thus fulfilling the requirement presented above. The α's are used as weights in Equation (2.27) to produce the PWCCA metric:

$$\text{PWCCA} \doteq \frac{\sum_{i=1}^{\tilde{r}} \alpha^{(i)} \rho_i}{\sum_{i=1}^{\tilde{r}} \alpha^{(i)}}. \tag{2.30}$$

2.4.2 Centered Kernel Alignment (CKA)

CKA [22, 23] builds upon the notion of dot product as similarity or alignment. It is constructed from the Gram matrices of the two representations, $G^{(1)} = A^{(1)}{A^{(1)}}^\top$ and $G^{(2)} = A^{(2)}{A^{(2)}}^\top$, both being $m \times m$ matrices containing the squared Euclidean norms between the m units in the two representations respectively. By denoting the flattening operator as "vec", we can compare the alignment of the two (flattened) Gram matrices using the dot product:

$$\mathrm{vec}\Big({G^{(1)}}^\top\Big)\mathrm{vec}\Big(G^{(2)}\Big) = \mathrm{vec}\Big(A^{(1)}{A^{(1)}}^\top\Big)^\top \mathrm{vec}\Big(A^{(2)}{A^{(2)}}^\top\Big). \tag{2.31}$$

The corresponding scalar can be thought as measuring the alignment between the two representations. Using the properties of the dot product and the Frobenius norm, we can analogously write Equation (2.31) as:

$$\mathrm{trace}\Big(A^{(1)}{A^{(1)}}^\top A^{(2)}{A^{(2)}}^\top\Big) = ||{A^{(2)}}^\top A^{(1)}||_F^2. \tag{2.32}$$

Assuming $A^{(1)}$ and $A^{(2)}$ are centered representations, the term ${A^{(2)}}^\top A^{(1)}$ can be thought of as the numerator of the cross-covariance matrix between ${A^{(1)}}^\top$ and ${A^{(2)}}^\top$—note that, for generic matrices X and Y centered w.r.t. their means and sharing row size n, we have that $\mathrm{COV}(X, Y) = X^\top Y/n^2$. Then, we can recover

$$\begin{aligned} ||\mathrm{COV}({A^{(1)}}^\top, {A^{(2)}}^\top)||_F^2 &= \left|\left|\frac{{A^{(2)}}^\top A^{(1)}}{n-1}\right|\right|_F^2 \\ &= \frac{\mathrm{vec}\Big({G^{(1)}}^\top\Big)^\top \mathrm{vec}\Big(G^{(2)}\Big)}{(n-1)^2} \\ &= \frac{\mathrm{trace}\big(G^{(1)}G^{(2)}\big)}{(n-1)^2} \end{aligned} \tag{2.33}$$

The quantity in Equation (2.33) is called *Hilbert-Schmidt Independence Criterion* (HSIC). It is applicable for *generic* Gram matrices, not necessarily those constructed from linear kernels as in Equation (2.31).

HSIC is not a relative index as it is based on covariance (and is also not invariant to isotropic scaling). It can be rendered invariant via normalization, which produces the CKA metric:

$$\begin{aligned} \mathrm{CKA} &\doteq \frac{\mathrm{HSIC}(G^{(1)}, G^{(2)})}{\sqrt{\mathrm{HSIC}(G^{(1)}, G^{(1)})} \cdot \sqrt{\mathrm{HSIC}(G^{(2)}, G^{(2)})} \\ &= \frac{\mathrm{trace}(G^{(1)} G^{(2)})}{\sqrt{\mathrm{trace}(G^{(1)} G^{(1)}) \cdot \mathrm{trace}(G^{(2)} G^{(2)})}} \end{aligned} \tag{2.34}$$

It was proposed as a metric for comparing neural representations by Kornblith et al. [63].

2.4.3 Normalized Bures Similarity (NBS)

NBS [13] is defined on Gram matrices of the representation, similarly to CKA, and it also shares all of its desided invariances:

$$\mathrm{NBS} \doteq \frac{\sqrt{\mathrm{trace}\Big(G^{(1)^{1/2}} \cdot G^{(2)} \cdot G^{(1)^{1/2}}\Big)}}{\sqrt{\mathrm{trace}(G^{(1)})\mathrm{trace}(G^{(2)})}}. \tag{2.35}$$

2.5 ML model compression

Model compression refers to the act of reducing the *memory footprint*, i.e., disk size or memory requirement, of a ML model. The necessity to *compress* a model may stem from drivers such as (a) reduce the energy consumption of the training or inference of the model, or (b) deploy the model on a resource-constrained device, such as a single-board computer or an embedded device. In the last decade, indeed, the astounding achievements attained by DNNs have gone hand in hand with their popularity and size: AlexNet, the first CNN for image classification to beat the ILSVRC challenge in 2012 [66] has around 62 million parameters. The Vision Transformer [30], another family of DNNs for Computer Vision developed in 2020, can have more than 600 million connections. Meanwhile, models for Natural Language Processing (NLP) are even larger: GPT-3 [11] has around 175 billion parameters,

while Wu Dao [92], released just one year later, boasts more than 1 trillion of them. This explosion of size in DNN-based models makes it extremely problematic to implement these tools in small machines or, even, as in the case of NLP, in personal computers or small computing clusters. This poses problems connected to the energy consumption and subsequent pollution caused by these models; moreover, it severely limits the access of advanced ML technologies to poorer countries, thus representing a burden to the democratization of Artificial Intelligence. Choudhary et al. [17] distinguishes between four different approaches for model compression:

- Pruning.
- Quantization.
- Knowledge distillation.
- Low-rank factorization.

While pruning and knowledge distillation are techniques for effectively reducing the number of parameters within the models, quantization and low-rank factorization leave the model's structure identical, but find ways to formulate or operate differently the computations inside the model such that they can be executed faster or using less memory, hopefully formulating predictions which are similar to the original ones. Within the context of this thesis, we will be expanding upon pruning and quantization, without going into the details of the other two techniques.

2.5.1 Pruning

Pruning indicates a family of techniques meant to reduce the complexity of parametric ML models by removing, according to predefined criteria, the number of parameters of said models, thus simplifying the relationship between the covariates and the response(s), drawing motivations and parallelisms to Occam's razor [123, 144]. The noun *pruning* originated from the tree-based decision models [10], as overgrown trees are prone to overfitting and, hence, the outer branches

need to be *pruned* in order to increase their generalization capability. Pruning is not limited to tree-based models, but extends generically to all parametric ML models. For instance, when working with linear regression, pruning is more often called *variable selection* [54], as removing parameters effectively equates to removing variables from the equation of the regression hyperplane. Pruning in the context of ANNs has been explored since the late '80s [55, 70] and a large body of works have been produced since then.

Hoefler et al. [50] categorize pruning techniques as (a) *data-driven* if the criteria for determining the parameters to be removed are dependent upon the model's response on a batch of data, or (b) *data-free* if the criteria for removal are strictly dependent upon only the model's architecture and parameters.

Additionally, we might have techniques operating pruning (a) *after training*, (b) *during training*—i.e., parameters are progressively removed while the model is being trained—, and (c) *before training*, i.e., the ANN is *sparsified* before any training is operated . In the cases (a) and (c), often a (re)training phase occurs as the pruned ANN is not able to formulate accurate predictions after the sparsification.

Finally, a last classification of ANN pruning techniques is operated according to the pruning criteria: (a) *unstructured pruning* occurs when the parameters are removed without regard for the geometry of the network, while (b) *structured pruning* techniques prune multiple parameters organized in *groups*. Examples of structured pruning include the removal of all of the synapses connected to a single neuron or the removal of one whole filter of a CNN. This discrimination is especially useful when motivating the choice for applying pruning. Indeed:

- Structured pruning produces a sparsity pattern in the parameters that can be immediately taken advantage of to reduce the dimension of the structure of the parameters. Given a layer l of a MLP having r_l neurons, let us assume to prune all of the connections incoming to and departing from the first neuron. This results in setting to zero the first column of the matrix of parameters W_l and the first row of W_{l+1}: these row and column can hence be removed from these matrices, leaving us with a dense, smaller

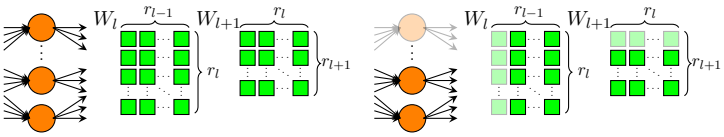


Figure 2.6: Effect of neuron pruning, a structured pruning technique, on a single layer of a MLP. The layer has r_l neurons, with $r_l \cdot r_{l-1}$ incoming connections and $r_l \cdot r_{l+1}$ outgoing connections, represented in the parameters matrices W_l and W_{l+1} respectively. Pruning, i.e., removing, the first neuron of the layer causes both the incoming and outgoing connections to be effectively deleted from the architecture of the MLP. Thus, the matrix W_l loses the first column, while W_{l+1} loses the first row. For simplicity, bias terms are omitted.

representation of the parameters. An illustration of this process is visible in Figure 2.6.

- Unstructured pruning is often producing sparsity patterns in the parameters that make it impossible to take advantage of the smaller number of parameters directly on the software side. Recently, a new set of Graphics Processing Units (GPUs), like the Nvidia Tesla A100[3], have been developed with support for sparsity on the matrix-matrix multiplication, thus allowing faster computation on hardware side.

Magnitude Pruning (MP)

In the context of this thesis, we experimented with both structured and unstructured data-free pruning techniques applied after training. The main approaches used fall under the categorization of MP, whereas the criterion for determining the parameters to remove is dependent upon the magnitude of the single parameter or the group of parameters to be pruned.

[3] `https://www.nvidia.com/en-us/data-center/a100/`

Unstructured MP (UMP) UMP works by ranking the parameters (of a layer, a groups of layers, or even the full ANN) depending on their absolute value, then removing the bottom $(\pi \cdot 100)\%$ of them, where $\pi \in [0, 1]$ is a hyperparameter called *pruning rate*. While implementing an unstructured pruning technique on a machine, due to the *irregular* nature of sparsity in the connectivity of the ANN which these technique produce, the structure of the parameters after pruning remains the same, with the pruned parameters *zeroed-out*. For instance, if we prune a MLP, the matrices of the parameters have the same dimensions after pruning, but the entries corresponding the pruned parameters remain at zero. This is conveniently implemented with the aid of a binary copy of the data structures of the parameters—called *mask*—indicating the status (pruned $= 0$, unpruned $= 1$) of each parameter. A pseudo-code implementation of UMP is shown in Algorithm 1. Notice that UMP can be applied *globally* (i.e., it can directly be applied to the whole structure of parameters of the ANN) or *locally* (i.e., it can be applied separately layer-by-layer, or to a group of layers or synapses), thus the algorithm takes as input a generic structure of parameters Θ which is represented, without loss of generality, as a vector.

The term *sparsity* is used to indicate the proportion of parameters which have been removed from an ANN. By recovering the mask M from Algorithm 1, the sparsity can be quickly calculated as $\frac{\sum_{i=1}^{p} M_{[i]}}{p}$.

Making UMP data-driven We defined UMP as a data-free method, acting on the basis of the value of the parameters, although it can be adjusted to be data-driven [88]. Instead of relying on the value of the parameters, this version needs the ANN to be evaluated on a representative dataset of size m. For each data point, each of the p synapses will carry a signal. We can store the signals corresponding to each data point and synapse in a matrix $V \in \mathbb{R}^{m \times p}$. In order to parallel the functioning of the data-free version of UMP, we can get the magnitude of the signals $V_{\text{abs}} = \text{abs}(V)$. Finally, we can rank the connections according to the mean absolute signal they carried, which

Algorithm 1: UMP applied to a generic set of parameters Θ.

input : Parameters Θ structured as a vector of dimension p; a pruning rate $\pi \in [0, 1]$.

output: The vector of parameters $\tilde{\Theta}$ after pruning; the pruning mask M.

$M \leftarrow \mathbf{1}_p$ `// mask as a vector of ones`
$\Theta_A \leftarrow \text{abs}(\Theta)$ `// absolute value of all parameters`
$\nu \leftarrow \pi$-th quantile of Θ_A
for $i \in \{1, \ldots, p\}$ **do**
    if $|\Theta_{[i]}| < \nu$ **then**
        `/* If magnitude of element of` Θ `is smaller than quantile, set the corresponding mask entry to 0 */`
        $M_{[i]} \leftarrow 0$
    end
end
$\tilde{\Theta} = \Theta \odot M$ `// Hadamard product`

we store in a vector $\Omega \in \mathbb{R}^p$:

$$\Omega = \frac{1}{m} \mathbf{1}_m^\top V_{\text{abs}}. \tag{2.36}$$

The vector Ω can effectively serve as a replacement of Θ_A in Algorithm 1: thus, the pruning methodology in this case becomes (a) get the mean absolute signal for each connection Ω, (b) get the π-th quantile of Ω, and (c) construct the mask M as indicated in Algorithm 1 by setting to 0 all positions whose corresponding value of Ω is smaller than the quantile. Throughout the present manuscript, unless otherwise stated, "UMP" will refer to the *data-free version*.

Structured MP (SMP) SMP requires first to define the concept of *magnitude* or *saliency* of the group of parameters to be removed.

For instance, within the context of CNNs, Li et al. [72] defined the saliency of a filter as the vector-L1 norm of the parameters within the filter. By recalling the notation introduced in Section 2.2.1, the parameters Θ_l for a generic layer l are stored in a 4D tensor of shape $Q_l \times Q_{l-1} \times \kappa_l \times \mu_l$, where

- Q_{l-1} is the number of channels of the input to layer l
- Q_l is the number of channels of the output of layer l
- κ_l and μ_l are respectively the height and width of the kernels of the cross-correlation.

The *filter* is defined as one of the Q_l 3D tensors of shape $Q_{l-1} \times \kappa_l \times \mu_l$ composing the parameters of the layer. The per-filter saliency $\omega_q^{(l)}$ is then defined as

$$\omega_q^{(l)} \doteq \sum_{i=1}^{Q_{l-1}} \sum_{j=1}^{\kappa_l} \sum_{k=1}^{\mu_l} \left| \Theta^{(l)}_{[q,i,j,k]} \right|, \quad q \in \{1, \dots, Q_l\}, \tag{2.37}$$

with $\Theta^{(l)}_{[\cdot,\cdot,\cdot,\cdot]}$ indicating a generic element of Θ_l. Thus, by drawing parallelisms with respect to UMP (Algorithm 1), we can sketch the implementation of L1 norm-based filter pruning, as shown in Algorithm 2. The immediate differences that we can notice with respect to UMP are:

- since structured pruning produces smaller parameters structures, instead of making use of a mask, we directly *remove* parameters from the original structure;
- while UMP is applicable globally or locally, Li et al. [72] suggest applying it only locally (i.e., specifically to a single convolutional layer).

Algorithm 2: SMP—L1 norm-based filter pruning for CNNs.

input : Parameters $\Theta^{(l)}$ of a generic convolutional layer of a CNN, a 4D tensor of shape $Q_l \times Q_{l-1} \times \kappa_l \times \mu_l$; a pruning rate $\pi \in [0, 1]$.

output: The vector of parameters $\tilde{\Theta}^{(l)}$ after pruning, its first dimension being $(Q_l - \lfloor \pi Q_l \rfloor)$.

$\Omega \leftarrow \mathbf{1}_{Q^{(l)}}$ `// vector for saliencies`
for $q \in \{1, \ldots, Q_l\}$ **do**
  $\Omega_{[q]} \leftarrow \text{saliency}(\Theta^{(l)}_{[q,\cdot,\cdot,\cdot]})$
end
$\Theta^{(l)}_A \leftarrow \text{abs}(\Theta^{(l)})$ `// absolute value of all parameters`
$\nu \leftarrow \pi$-th quantile of Ω
$\tilde{\Theta}^{(l)} \leftarrow$ `[ ] // empty structure for new parameters`
for $q \in \{1, \ldots, Q_l\}$ **do**
  if $\Omega_{[q]} \geq \nu$ **then**
    `/* If magnitude of` ω_q `is larger than quantile,`
    `the filter survives, else it gets removed */`
    $\tilde{\Theta}^{(l)}.\text{append}(\Theta^{(l)}_{[q,\cdot,\cdot,\cdot]})$
  end
end

The additional effect of filter pruning is that it does not only influence the dimension of the parameters of the current layer. The dimension of the parameters of the current layer is reduced from $Q_l \times Q_{l-1} \times \kappa_l \times \mu_l$ to $Q'_l \times Q_{l-1} \times \kappa_l \times \mu_l$, where $Q'_l \doteq Q_l - \lfloor \pi Q_l \rfloor$, i.e., we have removed

the $\lfloor \pi Q_l \rfloor$ pruned filters. Thus, the output of layer l is also smaller: $Q'_l \times h_l \times w_l$. This means also that the input to layer $l+1$ is smaller, thus also decreasing the dimension of its parameters, now being $Q_{l+1} \times Q'_l \times \kappa_l \times \mu_l$.

Retraining

We have mentioned before that pruned ANNs often require (re)training, as, after the pruning phase, the model performance in terms of accuracy is far from the one achieved by the denser counterpart. The literature distinguishes between different types of retraining, depending upon a number of factors, from which we mainly enucleate (a) number of epochs (b) learning rate schedule adopted, and (c) weight rewind strategy, the first two have previously been defined in Section 2.2.2. *Weight rewind* (WR) [33] specifies whether the surviving parameters of the ANN need to be rolled back to a previously-retained *initial state* before applying retraining. Techniques for WR will be presented in the next paragraphs.

Within the context of unstructured pruning, in practical implementations retraining is operated by incorporating the pruning mask in the training routine in order to "remember" which parameters are pruned and which are not.

Fine-tuning (FT) FT is a technique formalized by Han et al. [44] which involves retraining for a small amount of epoch, usually 1/100 to 1/10 of the epochs used in training the dense ANN, as suggested by them. Let us denote with λ_{fin} the value of the LR used during the latest training iteration. The retraining is operated by keeping a LR equal or smaller than λ_{fin}, motivated by the assumption that, despite some parameters having been pruned, the configuration of surviving parameters is still a good starting point for the pruned ANN, thus preferring exploitation to exploration. While usually inferior, in terms of accuracy, to other techniques which will be presented later [105], FT is still utilized as it offers a good compromise between accuracy and number of retraining epochs.

Weight Rewind (WR) WR, introduced in the context of the so-called "Lottery Ticket Hypothesis" (LTH) [33], introduces a *rewind* step before the retraining. After pruning, instead of directly retraining the ANN, the surviving parameters are rewound to a previous configuration, then the retraining is operated for the same number of epochs of the original training and with the same LR schedule. Initial works on the LTH suggested using the original parameters initialization of the dense ANN as the rewind configuration; it was later found, by Zhou et al. [145], that these configurations rendered retraining *unstable*. The authors then suggested using a configuration of parameters reached after few iterations or epochs of training.

Learning Rate Rewind (LRR) Building upon WR and FT, Renda et al. [105] later introduced LRR as a hybrid of the two. The authors noted that it was not necessary to rewind the weights to a previous configuration, provided that the same LR was used during retraining. They showed the superiority of LRR w.r.t. WR and FT with an extensive experimental protocol on several vision datasets.

Next, Algorithm 3 presents a pseudo-code implementation of UMP with WR retraining.

Iterative Pruning (IP)

Up to now, we have presented a pruning scheme which operates a single phase of synapses deletion with retraining. In literature, this methodology is also known as *one-shot pruning* (OSP). While OSP can be effective with small pruning rates (e.g., up to 0.5 or 0.6, depending upon the circumstances), it has been shown [33] that pruning should be iterated to reach higher sparsity levels. IP, though, was already known, albeit not experimented with, by LeCun et al. [70] in 1989. Indeed, the "prune + retrain" scheme previously presented can be seamlessly extended to introduce an iterative approach to pruning, where, after retraining, the model can be pruned again, then retrained, and so on until either (a) a given sparsity rate is reached, and/or (b) performance falls below a given threshold. This procedure is formally known as IP. A schematization of IP is depicted in Figure 2.7.

Algorithm 3: UMP with WR retraining.

input : Parameters Θ of a dense, non-trained ANN; pruning rate π; training epochs t; LR schedule $\mathcal{L}$; rewinding epoch $t_{\text{rewind}} \ll t$.

output: Sparse parameters Θ, pruning mask M.

if $t_{\text{rewind}} = 0$ **then**
  `/* If the rewind epoch is 0 (=initialization), store the initial parameters configuration. */`
  $\Theta_{\text{rewind}} \leftarrow \Theta$
end

`// Main training loop`
for $i \in \{1, \ldots, t\}$ **do**
  `/* Train one epoch of the ANN with given parameter configuration and LR schedule; optimizer omitted for simplicity. */`
  $\Theta \leftarrow \text{train\_epoch}(\Theta, \mathcal{L})$
  if $t_{\text{rewind}} = i$ **then**
    $\Theta_{\text{rewind}} \leftarrow \Theta$
  end
end

`// Prune and retrain`
$\Theta, M \leftarrow \text{UMP}(\Theta, \pi)$
for $i \in \{1, \ldots, t\}$ **do**
  `/* Retrain with mask to mark pruned parameters. */`
  $\Theta \leftarrow \text{train\_epoch}(\Theta, \mathcal{L}, M)$
end

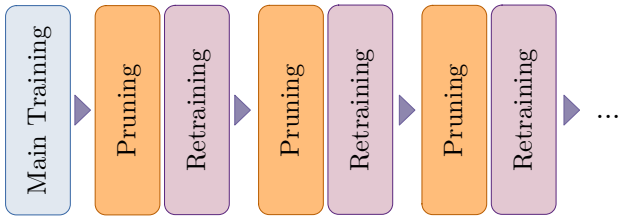


Figure 2.7: Schematization of IP. The process can go on arbitrarily until a user-defined stopping criterion is reached. "Main Training" refers to the original training of the dense ANN.

Frankle and Carbin [33] suggest that, in the context of UMP, IP be ran for many iterations, keeping in each a fixed small pruning rate while retraining with WR or LRR. The standard values that they use is 20 iterations of IP, each with a pruning rate of 0.2. The sparsity rate of the final model can be quickly computed as the complementary of the pruning rate of each iterations to the power of the number of iterations: $(1-0.2)^{20} = 0.8^{20} \approx 0.0115$, which means that the final model retains slightly more than 1% of the neurons of the dense ANN, with a sparsity which is slightly lower than 99%. We will indicate Iterative UMP with the acronym UIMP.

Throughout the current manuscript, we will use the following notation: "*pruning technique* + *retraining technique*". For instance, "UIMP + WR" indicates UIMP with retraining operated using WR.

2.5.2 Quantization

Quantization indicates the process of reducing the number of bits (*precision*) employed for the numerical representation of the parameters of a ML model. In modern-day ANNs, the parameters are represented using floating-point variables with 32 bits of precision [43], FP32 in short, which uniquely identify 2 147 483 647 unsigned rational numbers. According to the IEEE 754-2008 standard [20], FP32 can represent numbers in the interval $[-3.4 \cdot 10^{38}, 3.4 \cdot 10^{38}]$. Since it is impossible for a computer to represent *all* of the infinite rational and real numbers in

this interval, the floating-point encoding represents a fixed set of rational numbers, with a higher density the closer the number is to 0. While FP32 (and FP16) provide a good trade-off between accuracy and speed of inference [43], it can be sometimes necessary to further reduce the precision, for example when the device where the model needs to be deployed (or trained) has (a) limited memory, or (b) limited computational capability. This is the case of small devices, like single-board computers, e.g., Raspberry Pi's, or embedded/wearable devices. Running DNNs real-time on these machines can be an arduous task that can hinder the widespread adoption of these models in areas such as video-surveillance, smart health, or any application where factors like energy consumption do not allow for large computers—or even high performance computing clusters—to operate 24/7 to run these models. Quantization usually scales down a model to integer precision with 8 or even 4 bits (i.e., INT8 and INT4 encodings) to represent the parameters. Usually, quantization is applied *post-training*: the model is trained with FP32 precision and then scaled down to INT8 or INT4; it is also possible to directly train quantized models, although, in the context of this thesis, we will not treat this subject.

The most straightforward quantization is *rounding*: the real line is subdivided into equidistant intervals of length $\varepsilon \in \mathbb{R}$: this number is the *quantization step size*, and the numbers in the real line will be mapped to multiples of ε. The quantization, in this case, can be seen as an operator $q : \mathbb{R} \to \mathcal{B} \doteq \{n\varepsilon\}_{n\in\mathbb{N}}$. Given a real number x, q is defined as

$$q(x) = \begin{cases} \varepsilon \left\lfloor \frac{x}{\varepsilon} \right\rfloor & \text{if} \left\lfloor \frac{x}{\varepsilon} \right\rfloor \leq \frac{x}{\varepsilon} < \left\lfloor \frac{x}{\varepsilon} \right\rfloor + \frac{1}{2} \\ \varepsilon \left\lceil \frac{x}{\varepsilon} \right\rceil & \text{otherwise} \end{cases}$$

The same formulation can be used when generically converting numbers from different quantized representations. The final step which we need to consider is give an upper and lower bound to the output of q in order to render its output representable inside a machine. So, given $\mathcal{B} \ni \ell < u \in \mathcal{B}$, respectively lower and upper bounds of the quantization, we can extend q into producing the final quantization function:

$$\text{quant}(x) = \text{clip}(q(x), \ell, u),$$

where the clip operator is defined as:

$$\text{clip}(x, a, b) = \begin{cases} a & \text{if} x < a \\ b & \text{if} x > b \\ x & \text{otherwise} \end{cases},$$

with $a < b$. With time, other types of quantization have been proposed, like Vector Quantization [41], which produces representations which are denser around areas of the parameter space where the parameters are more clustered, although these methods fall outside the scope of the current thesis.

Applying quantization solely to the parameters of an ANN is useful only for reducing the size of the model, but does not automatically equate to a speed-up in inference time: in fact, the data still comes in FP32. In order to process the data, it is necessary to either (a) quantize the data to the same precision as the parameters, or (b) convert the parameters back to FP32. Of course, (a) is the preferred solution in terms of speed of inference. There exist multiple strategies for handling the quantization of data: TFLite [132], a ML environment specific for training and deploying ML models into low-end devices, proposes two solutions:

(a) *Full-integer quantization* (FIQ), which requires a preemptive step of *evaluation* of the data with a so-called *representative dataset*.

(b) *Dynamic-range quantization* (DRQ), which does not require the representative dataset, but quantizes the data on-the-fly.

FIQ is aimed at optimizing inference speed or for deploying the model on hardware which supports solely integer arithmetics. DRQ, on the other hand, can lead to higher accuracy than FIQ, but the hardware needs to support floating-point arithmetics; in addition, the dynamic quantization of the data can prove to be a large overhead, slowing down the execution of the model.

3 Experimental Investigations

In this chapter, we are going to present two work lines concerning experimental investigations on the phenomenon of pruning applied to DNNs trained using SGD-based techniques. In the first work, which is presented in Section 3.1, we compared hidden representations of pruned CNNs in order to find insights or regularities in the effect that pruning has on the representations. In the second experimentation, presented in Section 3.2, we work towards finding more efficient retraining schedules for UMP-based pruning schemes.

3.1 Comparing hidden representations of pruned and dense CNNs

The analysis of hidden representations learned by DNNs is an arduous task: the representations lie in a high-dimensional space and the data points composing these representations are embedded inside lower-dimensional manifolds, which is raveled inside the higher-dimensional space in complex manners, as introduced in Section 2.3. In addition, it is difficult to get a grasp on the semantic meaning underlying the representations learned by hidden layers due to the black box nature of ANNs, although we know that the manifold gets progressively unraveled as we progress layer through layer [3]. One line of the literature has concentrated on finding appropriate metrics suitable for operating comparisons between generic representations produced by (hidden) layers, a topic which we have covered in Section 2.4. Aside from the *classical* investigations on parallelisms between biological and ANNs [76] or comparisons between different architectures of DNNs [91, 99], such techniques have allowed for gaining insights on some specific properties of DNNs. For instance, Wu et al. [134] discovered that DNNs for NLP tasks show similar structures across different languages; Song and

Shmatikov [115] showed that overly-trained ANNs can learn to repeat training data, thus exposing the risk of leaking sensitive data; Mehrer et al. [83] discovered that DNNs with different initialization can lead to substantially different representations despite similar performance; finally, Frankle et al. [35] uncovered an early-phase of training in which ANNs are highly unstable to small perturbations.

In our investigations, we compared the representations learned by pruned CNNs with the ones produced by their unpruned counterpart, using the metrics previously introduced in Section 2.4. The rationale behind this investigation was connected to the possibility of uncovering possible trends, regularities, or insights in the pattern produced by the similarities. These factors could contribute in forming a deeper understanding on the mechanics underlying pruning or, more generally, training in DNNs. For our experiments, we considered two vision datasets: CIFAR10 [65] and Street View House Numbers (SVHN) [89]. We trained some CNNs on them, then we proceeded to prune them using UIMP, retraining with WR and LRR. After completing each retraining, we produced and saved the representations of each layer of the CNNs. Finally, for each layer, we operated the comparison of the representations of the pruned layer w.r.t. its unpruned counterpart. Our results were mixed, with CCA-based metrics and NBS giving one picture of the situation, and CKA producing a different one. In lieu of the higher credit given to CKA w.r.t. the other metrics [63], though, we came to a feeble conclusion that pruning seems to have an effect of progressive *detachment* of the representation learned by the dense CNN, with the final layers exhibiting a more pronounced dissimilarity.

Our experimentations have been published first in [4], then expanded upon in [5].

3.1.1 The datasets

As previously cited, we have trained our CNNs on two datasets for vision tasks, CIFAR10 and SVHN.

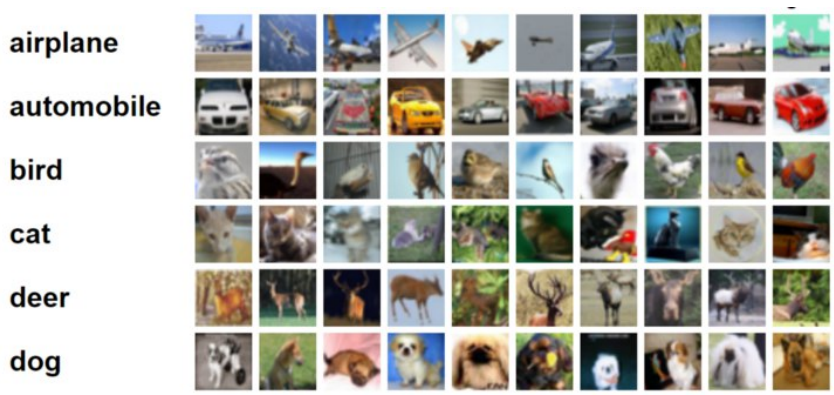


Figure 3.1: Sample of images from CIFAR10. Picture modified from [64].

CIFAR10

CIFAR10 is a dataset for image classification composed of 60 000 color images of size $3 \times 32 \times 32$. The images are equally split in 10 categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The categories encompass two different domain, 4 being means of transportation and 6 being animals. The dataset comes pre-split into a trainset of 50 000 data points and a test set of 10 000 images. A sample of the images from various categories is shown in Figure 3.1.

In [3], we experimented with creating a cascade of more complex problems out of CIFAR10. We thus extracted a subset of 2, 4, 6, and 8 categories from CIFAR10 and called the corresponding datasets "CIFAR2", "CIFAR4", "CIFAR6", and "CIFAR8". The categories composing this datasets are presented in Figure 3.2. The datasets were designed in order to be of non-trivial difficulty: CIFAR2 is composed of automobiles and trucks, which are similar means of transportation; CIFAR4 adds airplanes and ship; CIFAR6 introduces animal categories, *etc.*

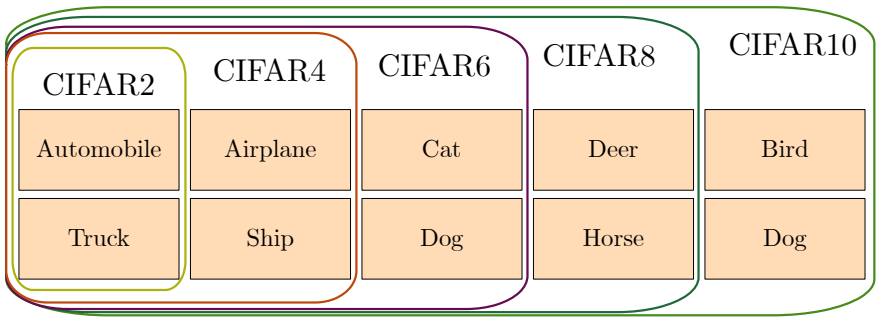


Figure 3.2: Categories composing the subsets of CIFAR10 which we used in [4].

SVHN

SVHN is composed of 99 289 color images depicting house numbers extracted from the Google Street View service of Google Maps[1]. The dataset is originally designed for object detection with the goal of having models learn to recognize and localize digits within images, even in situations in which there are multiple instances present in the same image. An image classification task has been designed for SVHN by having the images centered on one digit and cropped to the common spatial size of 32×32 (the same as CIFAR10). Thus, the task of models trained on this dataset is to classify the central digit in the 0-9 categories, ignoring the *distraction* provided by additional, non-central digits within some images. The dataset is already pre-split into a train-set of 73 257 images and a test-set of 26 032. A sample of images from SVHN is shown in Figure 3.3.

3.1.2 CNN architectures

In the present work, we made use of CNNs from two architecture families: VGG and ResNet, both having been introduced in Section 2.2.4. Specifically, for CIFAR10 we used some custom variants of VGGs, a

[1] `https://maps.google.com/`

Figure 3.3: Sample of images from SVHN. Picture from [119].

VGG16 (already presented in Section 2.2.4), and a custom ResNet variant, while for SVHN we have used another custom VGG.

Custom VGGs for CIFAR10 and SVHN

We first built four variants for fitting CIFAR10 and its smaller subsets introduced in Section 3.1.1. These variants were built progressively deeper and wider to tackle the increasing difficulty of the subsets. These models we called "VGG-1", "VGG-2", "VGG-3", and "VGG-4". Similarly, for SVHN we crafted another VGG-based CNN, slightly smaller than VGG11, which we called "VGG-SVHN". The structure of these CNNs is presented in Table 3.1. Recall that convolutional blocks, in the context of VGG, are blocks composed of a variable number of convolutional layers, after which max pooling is applied.

Table 3.1: Architecture of the five custom VGG CNNs VGG-1, VGG-2, VGG-3, VGG-4, and VGG-SVHN. Their structure is a variant of VGG (stack of convolutional blocks + MLP layers). Each convolutional block is written as {number of convolutional layers × convolutions per layer}. MLP layers instead are characterized only by their number of neurons. All convolutional and MLP layers use the ReLU activation function, except for the last MLP layer, which uses the softmax. The size of the final MLP layer, k, is dataset-dependent.

Dataset	Name	Conv. blocks	MLP layers
CIFAR	VGG-1	$\{2 \times 16\}$	256, k
	VGG-2	$\{2 \times 16\}; \{2 \times 32\}$	256; k
	VGG-3	$\{2 \times 64\}; \{2 \times 128\}; \{2 \times 256\}$	256; k
	VGG-4	$\{2 \times 64\}; \{2 \times 128\}; \{2 \times 256\}; \{2 \times 512\}$	1024; k
SVHN	VGG-SVHN	$\{2 \times 32\}; \{2 \times 64\}; \{2 \times 128\}; \{1 \times 256\}$	k

Custom ResNet for CIFAR10

The custom ResNet architecture we use for CIFAR10 was developed by Paige [94] and it is part of a collection of CNNs whose structure is optimized for fast training on CIFAR10 in the context of DawnBench [19]. DawnBench is a competition consisting in training ML models on a number of benchmark dataset using few resources and time—usually the focus is on training speed on a single desktop-level GPU, while ensuring a minimum level of test set accuracy. The architecture, whose schematic depiction is shown in Figure 3.4, differentiates itself w.r.t. the ResNets developed by He et al. [46] due to:

- the smaller number of residual blocks: this variant uses just 2 BasicBlocks, while ResNets in [46] always have 4 BasicBlocks or Bottlenecks, and
- the presence, between the two residual blocks, of two convolutional layers with subsequent max pooling and no skip connection; in [46], these kind of layers occur only at the beginning, in the so-called *preparatory layer*.

3.1.3 Training

As far as training is concerned, our two publications concerning this line of research ([4] and [5]) have substantial differences. We will distinguish between these two works by calling them "Early production" and "Later production". The common factor between these two is the fact that the models were trained using the CE loss introduced in Equation (2.3) and were all assessed according to their test-set accuracy.

Early production

In our early work, we trained the custom VGG variants (VGG-1 to VGG-4) up to a reasonable amount of final accuracy on CIFAR10 and its sub-datasets. In all cases, we used SGD with a batch size of 128 and the Adam optimizer presented in Section 2.2.2 with a LR of 0.001, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. In addition, we used L2-norm

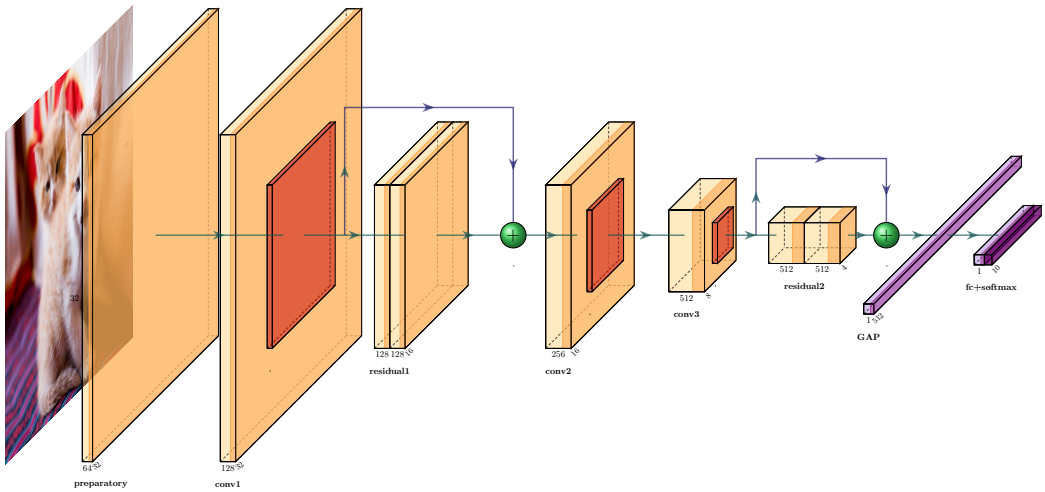


Figure 3.4: Architecture of the ResNet variant developed in [94] and optimized for fast training in CIFAR10. Convolutional layers are depicted in yellow, while max pooling is depicted in red, GAP and MLP layers are drawn in teal, the final softmax in gray. Activation functions are ReLU everywhere except for the final MLP layer, all convolutional layers include a BN step between the cross-correlations and the activation function. Numbers at the bottom of layers indicate the number of channels of the output of the same layer, while the number below the diagonal side indicates the spatial dimension (assuming $h = w$), supposing an initial image of size $3 \times 32 \times 32$.

regularization with a coefficient of 0.0005. Finally, we applied data augmentation during training time (thus re-rolling those augmentation routines employing stochastic hyperparameters at each new batch) applying (a) random cropping, (b) horizontal flipping with probability 0.5, and (c) roto-translation by a small random angle/shift. The test-set accuracy attained by these models was progressively decreasing with the complexity of the task, and ranged from 0.9103 to 0.8611. We present a summary of the training hyperparameters and the results in Table 3.2.

Later production

In our second manuscript, we trained the VGG16 and custom ResNet on CIFAR10 and the VGG-SVHN on SVHN.

The VGG16 was trained using SGD with momentum from Equation (2.14) with $\beta = 0.9$ and a batch size of 128. We also applied L2-norm regularization with a coefficient of 0.0001. We trained for 160 epochs using a step-wise decaying LR schedule, with a decrease of $0.1\times$ at epochs 80 and 120. We recorded a test-set accuracy of 0.9291.

We trained the VGG-SVHN using SGD with a momentum coefficient of 0.9, for 15 epochs using a step-wise decaying LR schedule, with a decrease of $0.1\times$ at epochs 7 and 12. The test-accuracy achieved was 0.9522.

Finally, we trained the ResNet using SGD with a batch size of 512 and a L2-norm regularization with a coefficient of 0.0001. As from [94], we employed LR schedule called LARS [139]. This schedule updates the LR at each iteration of training and is empirically tuned to work with high batch sizes—in the case of CIFAR10, 512 is considered a large batch size. The test-accuracy recorded by the ResNet was 0.9421. Table 3.2 summarizes hyperparameters and results of these models, along those of the earlier production.

3.1.4 Pruning

After training the CNNs trained in Section 3.1.3, we proceeded to prune them using UIMP. The retraining stratagem used was WR in the case

Table 3.2: Summary of the hyperparameters and the test-set accuracy for the CNNs in Section 3.1.3. The table logically separates the "early production" (upper half) and the "later production" (lower half). Legend: t: Number of epochs. LR=x ($\times m$ @ $t = t_1, t_2$): step-wise LR decay—initial LR of x multiplied by a factor m at epochs t_1 and t_2. β refers to the momentum coefficient from Equation (2.14). λ refers to the L2-norm regularization coefficient from Equation (2.4). BS: Batch Size.

CNN	Dataset	T	Optimizer [hyperparams.]	Test-set accuracy
VGG-1	CIFAR2	50		0.9103
VGG-2	CIFAR4	50	Adam [LR = 0.001, β_1 =	0.9008
VGG-3	CIFAR6	100	0.9, β_2 = 0.999, λ = 0.9,	0.8759
VGG-4	CIFAR8	100	BS = 128]	0.8744
VGG-4	CIFAR10	200		0.8611
VGG16	CIFAR10	160	SGD [LR=0.1 ($\times$0.1 @ $t = 80, 120$), $\beta = 0.9$, $\lambda = 0.0001$, BS = 128]	0.9291
VGG-SVHN	SVHN	15	SGD [LR=0.1 ($\times$0.1 @ $t = 7, 12$), $\beta = 0.9$, BS = 50]	0.9522
ResNet	CIFAR10	24	SGD [LR=0.1 (LARS schedule), $\lambda = 0.0001$, BS = 512]	0.9421

of the earlier production, LRR in the later one. The reason for this was that, at the publication of [4], LRR had not been released yet. For each model, we applied UIMP 20 times, each time starting from the same fully-trained dense CNN, this in order to get an average of multiple results concerning both accuracy and the values of the metrics for comparison.

UIMP + WR

WR was applied in its *late reset* formulation [34], by rewinding the weights to the configuration reached at the beginning of the third epoch. Pruning was applied locally, with the same outline as Zhou et al. [145]: they pooled convolutional layers and MLP layers in two separate groups and pruned them independently:

- Convolutional layers-only: pruning was applied with a rate of 0.1.
- MLP layers-only: pruning was applied with a rate of 0.2.

We repeated this scheme for the CNNs VGG-1, VGG-2, and VGG-3, since they globally have a very small number of parameters in the convolutional layers; concerning the CNN VGG-4, instead, we increased the pruning rate of the convolutional layers to 0.2, while still pruning the two pools independently. We excluded BN layers from pruning.

As for the indications of Frankle and Carbin [33], we performed UMP for 20 iterations; in all cases, the final sparsity was between 98.5 and 99.0%.

UIMP + LRR

In [5], we instead shifted to pruning using UIMP + LRR. The VGG16, VGG-SVHN, and ResNet were pruned for 20 iterations, using global pruning instead of the pooled variant previously mentioned. Again, we did not prune BN layers. The pruning rate at each iteration was 0.2 for all types of layers, thus producing pruned CNNs with an overall sparsity of 0.9885.

3.1.5 Producing the neural representations

We then proceeded to obtain the neural representation for all the layers in the CNNs. We computed the representations using a random sample of $m = 5000$ datapoints from the trainset. The layers interested by this analyses were the following:

- Convolutional layers *after* the application of the activation function: we will indicate them, in short, `conv`; `conv_res` in case of convolutional layers belonging to a residual block.
- Max Pooling and Average Pooling layers, in short, `pool`.
- Addition node for skip connections (applicable only in the case of ResNet), in short, `add`.
- Only in the early production, MLP layers, in short `f-c`.
- Only in later production, output layers, in short `out`, before the application of the softmax.

We obtained the representations for all these layers for each fully-trained CNN after the first, dense training, and after each retraining. So, for each CNN and layer, we had 21 representations: one *dense*[2], all the others *pruned* at various degree of sparsity.

3.1.6 Comparing the representations

In our earlier production, we compared the representation using Mean SVCCA Similarity; later, given a new score of work published on the matter of similarity metrics for neural representations [63, 122], we expanded our research by incorporating PWCCA, CKA, and NBS.

We operated an *intra-layer* comparison, i.e., we compared each layer with itself at various steps of UIMP. Specifically, we calculated the similarity between the *dense* version of the representation and the *pruned* versions. Let us denote the representation of a layer l at the iteration

[2]Notice that *dense*, in this case, refers to the parameters space. The representations are unlikely to present sparsity due to pruning, but rather due to the ReLU activation function.

of pruning j as $A^{(l,j)}$. In our case, j goes from 0 to 20, and $j = 0$ indicates the 0-th iteration of UIMP, i.e., the training of the unpruned CNN. Thus, for each layer l and metric s, we had 20 values of similarity: $s(A^{(l,0)}, A^{(l,1)}), \ldots, s(A^{(l,0)}, A^{(l,20)})$.

For what concerns layers whose representation had a 4D structure (e.g., convolutional layers), in the case of SVCCA, and PWCCA we operated the reshaping to matrix according to Equation (2.29); for the other metrics, according to Equation (2.28).

3.1.7 Results

Test set accuracy

First, we will quickly give an overview on the test set accuracy of the models through the various iterations of pruning. Notice that the goal of this work is the comparison of CNN layers, not the training of state-of-the-art CNNs concerning accuracy. We nonetheless concentrated on producing well-performing models. As indicated in Section 3.1.4, we operated UIMP for 20 iterations, and we repeated this procedure 20 times per CNN, starting from the same dense, fully-trained model. We then averaged the test set accuracy per CNN and pruning iteration, reporting the median value. These results are showcased in Figure 3.5 for the "early production" and in Figure 3.6 for the "later production". We can notice that:

- For all the datasets derived from CIFAR10, the test set accuracy of the pruned models remains roughly on-par with the dense counterpart.
    - The custom ResNet, architectonically optimized on this dataset, is able to get very high performances, and can reach 95% of accuracy in some instances with moderate to high pruning (sparsity roughly between 0.3 and 0.9).
- The same cannot be said for SVHN: pruning seems to be detrimental as it causes a seemingly linear decrease in accuracy, from more than 95% to less than 94%.

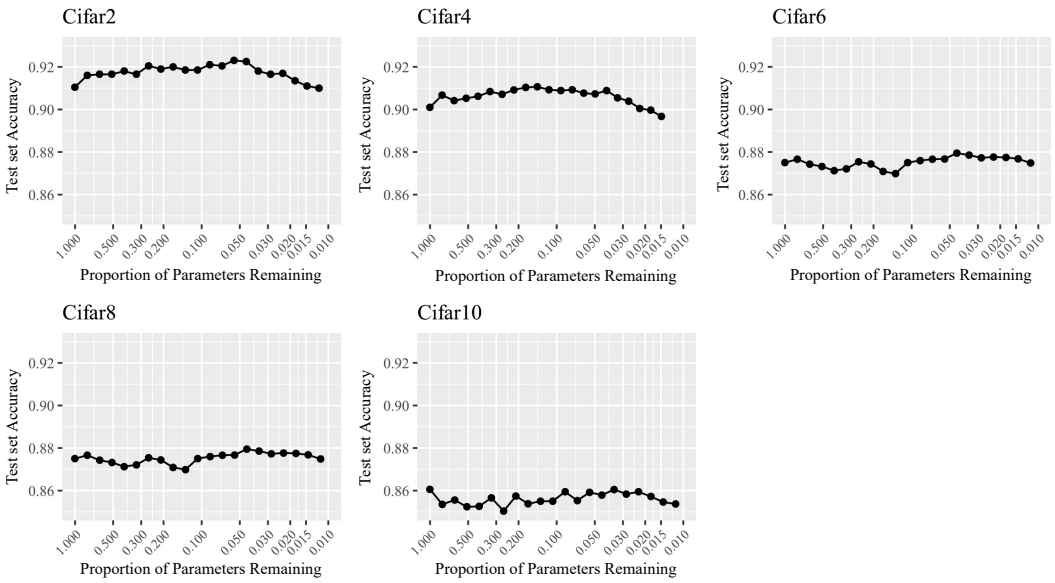


Figure 3.5: Median test set accuracy for each of the CNNs of the "early production" on the datasets CIFAR2-10. VGG-1 was trained on CIFAR2, VGG-2 on CIFAR4, VGG-3 on CIFAR6, VGG-4 on CIFAR8 and CIFAR10. The performance of the dense model is at proportion of parameters remaining = 1. Note: proportion of parametersremaining = (1 − sparsity). x axis shown in logarithmic scale for the sake of visualization.

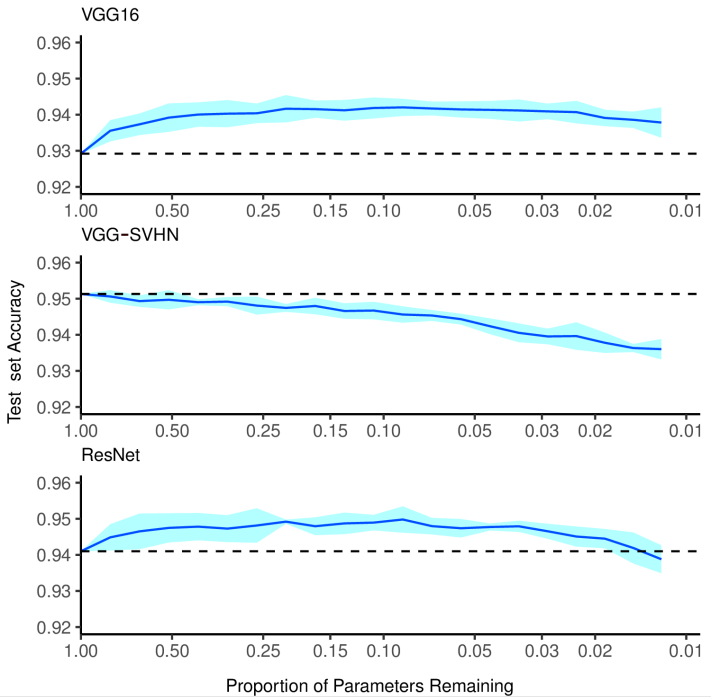


Figure 3.6: Median test set accuracy and error bands for the VGG16 (trained on CIFAR10), VGG-SVHN (trained on SVHN), and ResNet (trained on CIFAR10). The performance of the dense model is at proportion of parameters remaining = 1 and is represented by the dotted line. Note: proportion of parameters remaining = (1 − sparsity). x axis shown in logarithmic scale for the sake of visualization.

Similarity metrics

Next, we calculated the similarity as explained in Section 3.1.6. We computed Mean SVCCA Similarity on the representations of VGG-1, VGG-2, VGG-3, and VGG-4; for VGG16, VGG-SVHN, and ResNet we computed Mean SVCCA Similarity, PWCCA, CKA, and NBS. Having at disposal 20 representations per CNN and UIMP iteration, we calculated the metrics for each of these and then averaged them, reporting the median.

The charts in Figures 3.8 to 3.10 showcase a different picture depending upon the metric used. While Mean SVCCA Similarity and PWCCA have a distinctive "U" shape, this is rather absent in CKA and NBS. The "U" shape is also present for the smaller CNNs from the early production (Figure 3.7) for the datasets CIFAR6-10. This trend led us to formulate a belief that the initial and final layers of a DNN, trained on a sufficiently complex problem, were the most important in the context of pruning, being the representations they produced similar. This hypothesis would look logical: visual low-level features are usually similar across different problems [90] and, in order to perform similarly on the same data, two models should learn similar higher-level features (hence, similar representations for the *deepest* layers); thus, pruned CNNs were working similar solution from different configurations of mid-level features, with the difference being more pronounced as the pruning is more aggressive. CKA (and, on a minor note, NBS) seems instead to provide a different picture: low-level features are still similar across pruning iteration (especially on CIFAR10), but we see a slight decrease in similarity as we progress through deeper layers. Also, as observed before, the more we prune, the more different the representations look. Given the seemingly higher reliability that CKA has been credited with [63, 122], we then proceeded to set aside our original hypothesis, concluding that pruning does instead seem to cause a progressive detachment between the features learned in the dense vs. the pruned models. We also tried with computing CKA and NBS according to the reshaping from Equation (2.29), but the results were unchanged.

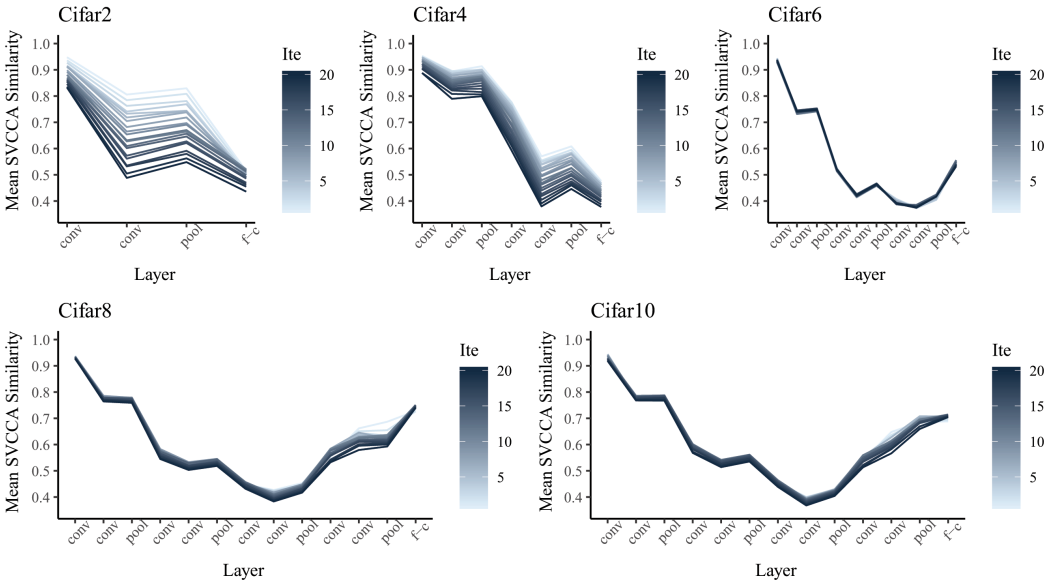


Figure 3.7: Median value of Mean SVCCA Similarity for each of the CNNs of the "early production" on the datasets CIFAR2-10. VGG-1 was trained on CIFAR2, VGG-2 on CIFAR4, VGG-3 on CIFAR6, VGG-4 on CIFAR8 and CIFAR10. Color brightness of the lines indicate pruning iteration ("ite" in the legend): the darker the shade, the higher the iteration.

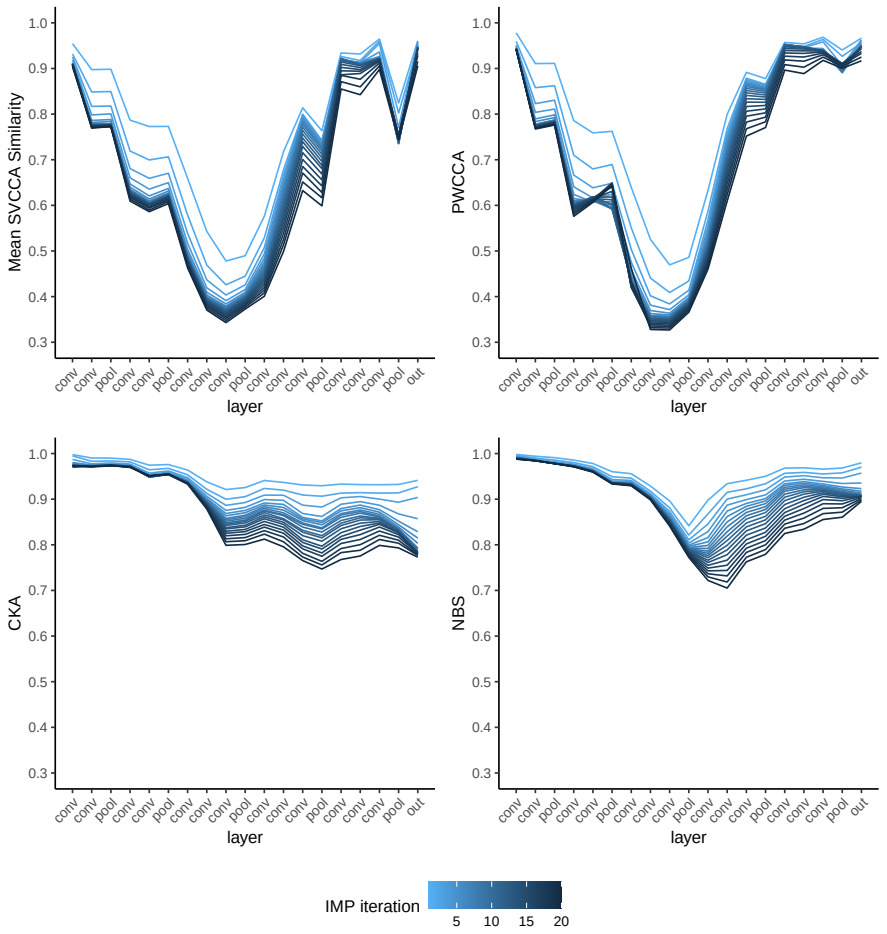


Figure 3.8: Median value of Mean CCA Similarity, PWCCA, CKA, NBS for the VGG16 trained on CIFAR10. Color brightness of the lines indicate pruning iteration ("IMP iteration" in the legend): the darker the shade, the higher the iteration.

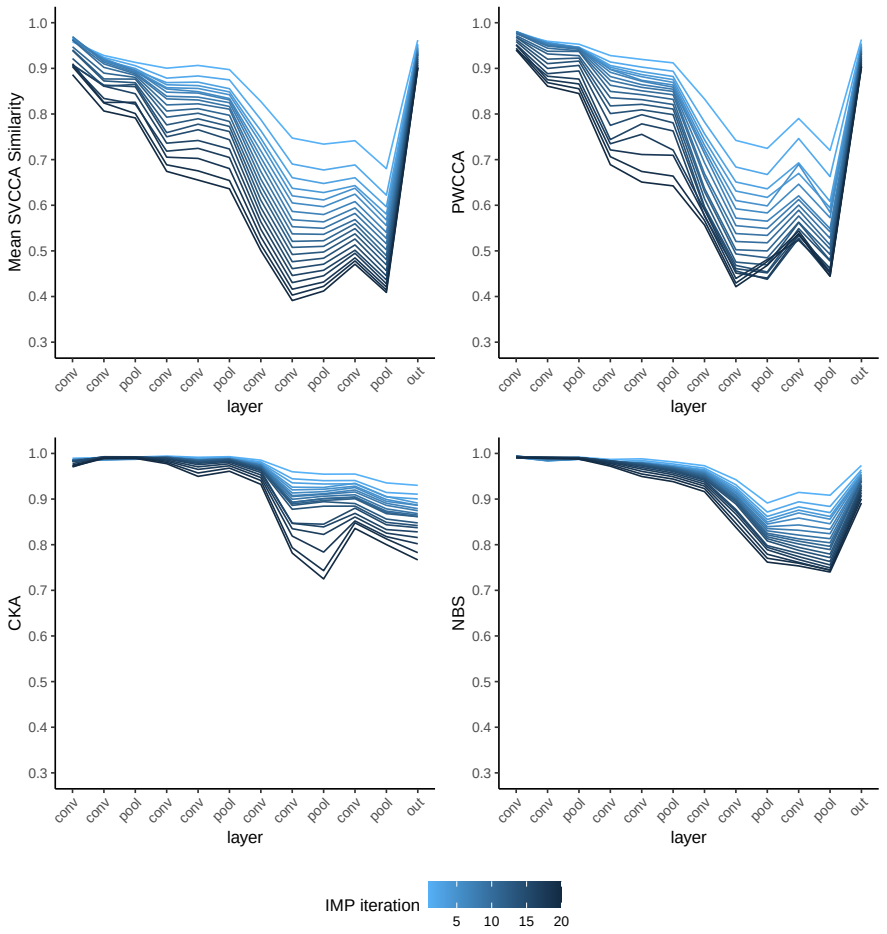


Figure 3.9: Median value of Mean CCA Similarity, PWCCA, CKA, NBS for the VGG-SVHN trained on SVHN. Color brightness of the lines indicate pruning iteration ("IMP iteration" in the legend): the darker the shade, the higher the iteration.

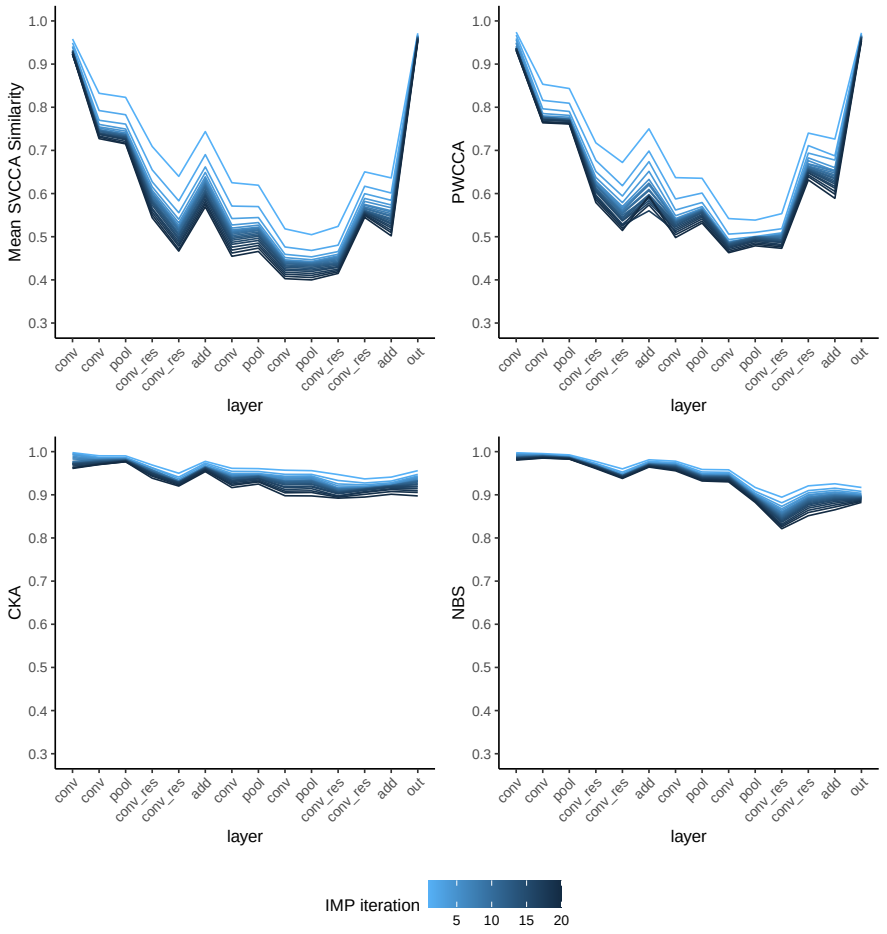


Figure 3.10: Median value of Mean CCA Similarity, PWCCA, CKA, NBS for the ResNet trained on CIFAR10. Color brightness of the lines indicate pruning iteration ("IMP iteration" in the legend): the darker the shade, the higher the iteration.

3.1.8 Additional notes on the similarity metrics

We concluded [5] by providing two critical commentaries on the invariance properties (Section 2.4) of the metrics for comparing neural representations:

- *Convolutional layers*: we argued that the rotational invariance requirement is excessive for comparing convolutional layers, as the outputs of these layer abide to specific spatial constraints of the 2D geometry of the image. This constraint is related to the way the cross-correlation is performed, which yields an inherent property of *spatial locality*, thus rendering invalid the requirement of rotational invariance. The invariance should be instead on the permutation of the channel dimension, as it is easy to permute the order of the channels in each layer and obtain the same output from the CNN. This invariance is easily enforced by performing the reshaping in Equation (2.29), which should hence *always* be employed when comparing convolutional layers.

- *Output layers*: also in this case the rotational invariance is excessive, as each neuron in an output layer has always the same meaning and, in classification, is related to the category corresponding to each neuron. Thus, in output layers, other metrics, like cosine similarity, can be used for a more accurate comparison. In Figure 3.11 we showcase the application of cosine similarity along with the other metrics for comparing the representation of the output layer. Given two representations A, B, we have:

$$\text{cosine similarity}(A, B) = \frac{\text{vec}(A)^\top \text{vec}(B)}{||A||_2 \cdot ||B||_2} \tag{3.1}$$

  Indeed, the cosine similarity shows a different behavior w.r.t. CCA-based metrics and NBS, while CKA exhibits a related pattern.

3.1.9 Conclusions and ensuing developments

In [4] and [5], we investigated the results of various similarity metrics (Mean SVCCA Similarity, PWCCA, CKA, NBS) applied to different

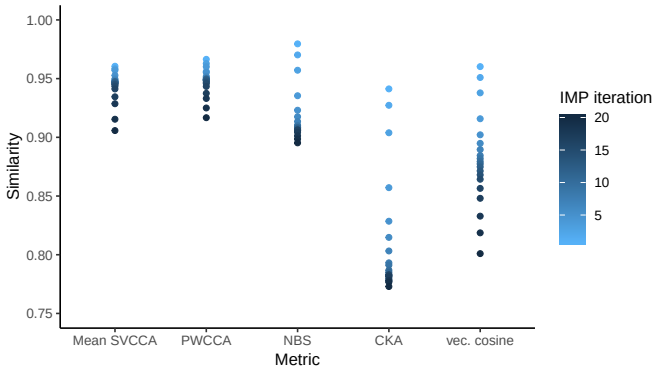


Figure 3.11: Detail of the similarity between representations for the output layer of the VGG16. "vec. cosine" is the cosine similarity from Equation (3.1).

variants of pruned layers of CNNs. We trained several CNNs on the datasets CIFAR10 and SVHN. We pruned them using UIMP from Section 2.5.1. Our goal was to uncover possible trends or regularities in the pattern of similarities for gaining insights on the inner working of pruning. Despite initial promising results [4], we did not, eventually, notice specific trends, concluding that pruning causes effectively a "progressive detachment" of the learned representations as pruning is performed.

Ensuing considerations

The matter of similarity for comparing neural representations has progressed in the two years from our last investigation on the matter: more specifically, CKA, despite still enjoying a widespread usage in the literature, has been partially discredited [24, 25, 28] and its effectiveness is yet to be proven on solid grounds. In the meanwhile, other statistics-based [125] and topology-based [7, 29] similarity metrics have been developed, as are rectified variants of CKA [24] which could lead the way to additional investigations.

3.2 Speeding-up UIMP

Frankle and Carbin [33] and later Renda et al. [105] refined the concept of UIMP [70], the former introducing the retraining scheme of WR, the latter presenting LRR. LRR is, as of today, still treated as a state-of-the-art retraining procedure, which is able to produce ANNs which, even at very high levels of sparsity, are competitive in terms of accuracy with the dense model. Despite this, the application of LRR or WR is also expensive in terms of both energy and time, as, starting from a dense model, this needs to be pruned for many times with a low pruning rate, and each time retrained for the same amount of epochs as the dense model. In the last few years, a lot of attention has been posed to the high energy consumption of training DNNs (and in some cases even of inference), both by the scientific literature [38, 95] and by the mainstream media [68, 124]. This can also open the door for considerations on pollution in the case the machines on which training/inference is performed do not use energy from green source.

UIMP, especially if coupled with WR or LRR retraining, can lead to a very large total training time (i.e., the total time it takes to get from the untrained dense model to the fully-trained pruned model at the last iteration of UIMP): if a dense DNN is trained in t_{train} epochs, adding pruning and retraining for t_{prune} times yields a total training time north of $(t_{\text{train}}+1)\cdot t_{\text{prune}}$, taking into account that the pruning phase requires some computation, albeit small w.r.t. training, and that the subsequent training phases are probably going to be more expensive as the pruning mask needs to be considered during training. Notice that (a) we do not consider SMP in the calculation above, as each iteration will effectively yield a smaller DNN, which will speed up training time, and (b) we assume training is performed on standard hardware that cannot take advantage of generic sparsity: also in this case, each retraining phase will be faster due to there being more sparsity. Thus, methods and heuristics that can accelerate IMP without compromising too much the final accuracy were and are the topic of researches still today. In 2020, we proposed Accelerated Iterative Magnitude Pruning (AIMP) [146], a meta-heuristic for speeding-up the execution of IMP by simply reducing the number of training iterations in some retraining phases.

As a meta-heuristic, AIMP can be applied generically to all pruning techniques consisting in more than one retraining iteration—e.g., UIMP and structured IMP, with any retraining paradigm. We empirically showed that, on vision tasks, AIMP + WR was consistently beating IMP + WR at high sparsity across different parameters configurations, and that AIMP + LRR was producing competitive results w.r.t. IMP + LRR.

3.2.1 Recall of UIMP and retraining paradigms

Recalling from Section 2.5.1, UIMP is a pruning paradigm which, given a dense DNN trained for t_{train} epochs, requires this model to be pruned (using UMP) and retrained for the same t_{train} epochs for a number of iterations, which we denote with t_{prune}. Each iteration, the DNN is pruned by the same pruning factor π. Usually, this number of iterations is known in advance and is determined by the target pruning rate desired. For instance, Frankle and Carbin [33] propose having $t_{\text{prune}} = 20$ and $\pi = 0.2$.

The retraining paradigm can be selected from FT, WR, and LRR, all introduced in Section 2.5.1. The first has been noted to perform poorly in conjunction with IP, while the latter two have been noted to produce highly-sparse and well-performing DNNs. Supposing a generic LR schedule $\mathscr{L}$ with final LR λ_{fin}:

- FT [44] retrains for a small amount of epochs $t_{\text{FT}} < t_{\text{train}}$, using a small LR $\lambda_{\text{FT}} \leq \lambda_{\text{fin}}$.
- WR [33] retrains for the same number of epochs t_{train} applying the same LR schedule $\mathscr{L}$, but requires, after each pruning phase, to *reset* the surviving parameters to their initialization Θ_0.
    - In addition, Zhou et al. [145] proposed using the configuration of parameters after a few training iterations of the dense DNN, $\Theta_{t_{\text{rewind}}}$.
- LRR [105] retrains for the same number of epochs t_{train} applying the same LR schedule $\mathscr{L}$ without resetting the parameters.

3.2.2 AIMP

AIMP builds on UIMP + WR and LRR by proposing to *cut short* the training of the *intermediate* iterations of IMP, i.e., those from the first to the penultimate one. Recalling that t_{train} is the number of training epochs of the dense DNN, we define $\tau \ll t_{\text{train}}$ and propose to retrain for only τ epochs in those intermediate iterations. We call τ *partial training epochs*. The immediate observation is that AIMP is a *meta-heuristic*, in the sense that it does not need to necessarily be tied to any pruning technique or retraining paradigm, the only two requirements are (a) the pruning technique must be iterative, and (b) the retraining phase must be comparable, in terms of epochs, to the training of the dense model. We, though, concentrated on applying AIMP on UIMP + WR or LRR. For this reason, from now on, unless otherwise stated, we will imply AIMP to be used with UIMP: thus, AIMP + WR will mean "AIMP used with UIMP + WR as a retraining paradigm". In addition, when referring to AIMP applied with a specific value of τ, we will refer to it as "AIMP$_\tau$". Algorithm 4 presents AIMP + LRR.

3.2.3 Dataset and CNN architecture

For this work, we trained various instances of VGG19, a VGG-based CNN, on the dataset CIFAR10, which we previously presented in Section 3.1.1. Regarding VGG, we have already extensively written about it in Section 2.2.4 and Section 3.1.2. With reference to VGG16, which is depicted in Figure 2.2, VGG19 adds one convolutional layer to each of the final three convolutional blocks, while keeping the hyperparameters of the layers the same. As previously done in Section 3.1.2, also in this case we use the "modern" variant, without hidden MLP layers and with BN after each convolution. In all cases, we use the ReLU activation function after both convolutional and MLP layers, with the sole exception being the output layers, where softmax is used.

3.2.4 Experimental settings

We operated two sets of experiments. The first one, which is also the bulkiest, operates a comparison between AIMP + WR by varying

Algorithm 4: AIMP + LRR.

input : Parameters Θ of a dense, non-trained ANN; pruning rate π; training epochs t_{train}; pruning iterations t_{prune}; partial training epochs τ; LR schedule $\mathcal{L}$.

output: Sparse parameters Θ and pruning mask M.

```
// Main training loop
for i ∈ {1, ..., t_train} do
    Θ ← train_epoch(Θ, L)
end
// Pruning iterations
for j ∈ {1, ..., t_prune} do
    // Determine # of epochs for current retraining
    t_current = τ
    if j = t_prune then
        // Last iteration = full training
        t_current = t_train
    end
    // Prune and retrain
    Θ, M ← UMP(Θ, π)
    for i ∈ {1, ..., t_current} do
        /* Retrain using the same LR schedule and mask
           to mark pruned parameters.                    */
        Θ, M ← train_epoch(Θ, L, M)
    end
end
```

several hyperparameters of retraining and pruning. The second one considers instead AIMP + LRR. Unless otherwise specified, the initial training for all these experiments was operated by training for a number of epochs t_{train} of 160, using SGD with a momentum of 0.9, L2-norm regularization of 0.0001, and a stepwise LR decay schedule, which decreases the LR by a factor of 10 at epochs 80 and 120.

AIMP + WR

When applying AIMP with WR, we devised four experiments, in each of these controlling a different hyperparameter to verify the effectiveness of it in determining the success of AIMP. In all experiments, UIMP + WR acts as a *control* to compare the results of AIMP. Unless otherwise stated, the hyperparameters of UIMP are the same as AIMP, with the sole obvious exception of τ: UIMP, in fact, always retrain for the same number of epochs t_{train}.

The experiments were the following:

(i) *Controlling* τ: we varied $\tau \in \{20, 30, 40, 50\}$, while keeping fixed $t_{\text{prune}} = 20$ and $\pi = 0.2$.

(ii) *Controlling* t_{train} *for AIMP only*: we tried limiting the number of epochs for training the dense CNN in AIMP to $\tau = 50$. For AIMP, then, only in the final pruning iteration we trained for all the 160 epochs. We kept fixed fixed $t_{\text{prune}} = 20$ and $\pi = 0.2$.

(iii) *Controlling* π: we varied $\pi \in \{0.2, 0.3, 0.4\}$, while keeping fixed $\tau = 50$. In order to produce ANNs with similar sparsity, we consequently varied t_{prune}: it is 20 with $\pi = 0.2$, 12 with $\pi = 0.3$, and 9 with $\pi = 0.4$.

(iv) *Controlling* t_{prune}: we varied $\pi \in \{3, \ldots, 20\}$, while keeping fixed $\pi = 0.2$ and $\tau = 50$.

AIMP + LRR

We applied AIMP to LRR with the hyperparameters $t_{\text{prune}} = 20$, $\tau = 50$, and and $\pi = 0.2$, providing comparison w.r.t. UIMP + LRR with

the same configuration. The reason for the limited experimentation with LRR stems from the fact that the work by Renda et al. [105], which introduced this technique was just published as we were completing our analyses; hence, LRR was a late addition to our work.

3.2.5 Results

With reference to the five sets of experiments presented in Section 3.2.4 (four with AIMP + WR, one with AIMP + LR), we performed five runs per set. The models pruned with AIMP were evaluated according to two different metrics: (a) test set accuracy, and (b) *speed-up* w.r.t. UIMP, calculated as

$$\text{speed-up} = \frac{\text{total training epochs UIMP}}{\text{total training epochs AIMP}}.$$

For instance, in setting (i), we have

$$\text{speed-up} = \frac{(t_{\text{prune}} + 1)t_{\text{train}}}{\tau(t_{\text{prune}}) + 2t_{\text{train}}} = \frac{160 \cdot 21}{\tau \cdot 19 + 2 \cdot 160}.$$

The results for all experiments are presented in Table 3.3. The summary of the results are:

Experiment (i): AIMP + WR seems to work well with various levels of τ: there seems to be a sligth decrease in accuracy with $\tau \leq 30$, but the difference with UIMP + WR is still negligible. On the other hand, the speed-up is large (starting from 2.64× with $\tau = 50$).

Experiment (ii): AIMP + WR seems to work even better when also the first training iteration is *cut short* at $t_{\text{train}} = \tau = 50$, with an even higher speed-up.

Experiment (iii): the success of AIMP + WR is not tied to the pruning rate; provided the final sparsity is high (in all cases, we got a final sparsity between 0.98 and 0.99), the pruned model performed roughly on-par with UIMP + WR.

Experiment (iv): this experiment gives a more complete overview on the functioning of AIMP; the accuracy of AIMP is well worse than UIMP at low sparsity, while it gets practically the same at sparsity ≥ 0.98.

AIMP + LRR: also in this case, AIMP seems to produce competitive results w.r.t. UIMP.

Next, we are going to present two charts that can help in shedding light on the functioning of AIMP. Figure 3.12 presents the outcome of experiment (i). We can see how the test set accuracy of the pruned network (after retrain) is well lower w.r.t. UIMP for most of the time; however, after the final iteration, the accuracy spikes at the level of UIMP. This is expected, since during the first 19 iterations of AIMP we train for a small amount of epochs. In Figure 3.13 we can instead appreciate the test set accuracy of AIMP and UIMP considering $t_{\text{prune}} \in \{3, \ldots, 20\}$. We can see how AIMP initially has some difficulties catching up with UIMP (even if, in the last iteration, we train for the full 160 epochs): after $t_{\text{prune}} = 8$, AIMP stabilizes around an accuracy of ~ 0.914, while UIMP starts falling; at $t_{\text{prune}} = 18$ they catch up, then produce roughly the same results, with AIMP being slightly better at $t_{\text{prune}} = 20$. This is indicative of the fact that AIMP is not inherently working on-par with UIMP, but does so only at high sparsities. If one wants to produce a highly-sparse ANN, then AIMP might effectively be a useful tool, being 200-300% faster than UIMP.

3.2.6 Analysis

Despite the promising results, we could not find a strong indication suggesting a potential candidate for τ. For the models trained during experiment (i), we analyzed:

- The *potential* pruning masks produced at each epoch of training. The term *potential* is used because pruning is not operated after each epoch. We then checked for differences in the masks produced at consecutive epochs, in order to see if there were notable shift in the masks. This can be an indicator of *pruning*

AIMP + WR

Experiment (i): $\tau \in \{20, 30, 40, 50\}$

τ	Sparsity	acc_{AIMP}	acc_{UIMP}	Δ_{acc}	Speed-up
20	0.9885	0.9001	0.9064	-0.0063	4.80
30		0.9039		-0.0025	3.78
40		0.9082		+0.0018	3.11
50		0.9071		+0.0007	2.64

Experiment (ii): $t_{\text{train}} = 50$ for first epoch of AIMP

t_{train}	Sparsity	acc_{AIMP}	acc_{UIMP}	Δ_{acc}	Speed-up
50	0.9885	0.9110	0.9064	+0.0046	2.90

Experiment (iii): $(\pi, t_{\text{prune}}) \in \{(0.2, 20), (0.3, 12)(0.4, 9)\}$

π	Sparsity	acc_{AIMP}	acc_{UIMP}	Δ_{acc}	Speed-up
0.2	0.9885	0.9071	0.9064	+0.0007	2.64
0.3	0.9862	0.9053	0.9081	-0.0028	2.39
0.4	0.9899	0.8998	0.8999	-0.0001	2.22

Experiment (iv): $t_{\text{prune}} \in \{3, \ldots, 20\}^{(*)}$

t_{prune}	Sparsity	acc_{AIMP}	acc_{UIMP}	Δ_{acc}	Speed-up
3	0.4880	0.8989	0.9281	-0.0292	1.52
6	0.7379	0.9082	0.9306	-0.0224	1.96
9	0.8322	0.9143	0.9285	-0.0142	2.15
18	0.9820	0.9140	0.9138	+0.0002	2.60
19	0.9856	0.9096	0.9093	+0.0003	2.62
20	0.9885	0.9071	0.9064	+0.0007	2.64

AIMP + LRR

τ	Sparsity	acc_{AIMP}	acc_{UIMP}	Δ_{acc}	Speed-up
50	0.9885	0.9362	0.9368	-0.0006	2.64

Table 3.3: Results of the experiments with AIMP + WR and AIMP + LRR. Legend: $\text{acc}_{\text{AIMP}}, \text{acc}_{\text{UIMP}}$: test set accuracy for the models pruned with AIMP and UIMP respectively; Δ_{acc}: difference between acc_{AIMP} and acc_{UIMP}; (*): only the most important results are shown, refer to Figure 3.13 for more details on the selection of the values of t_{prune} shown.

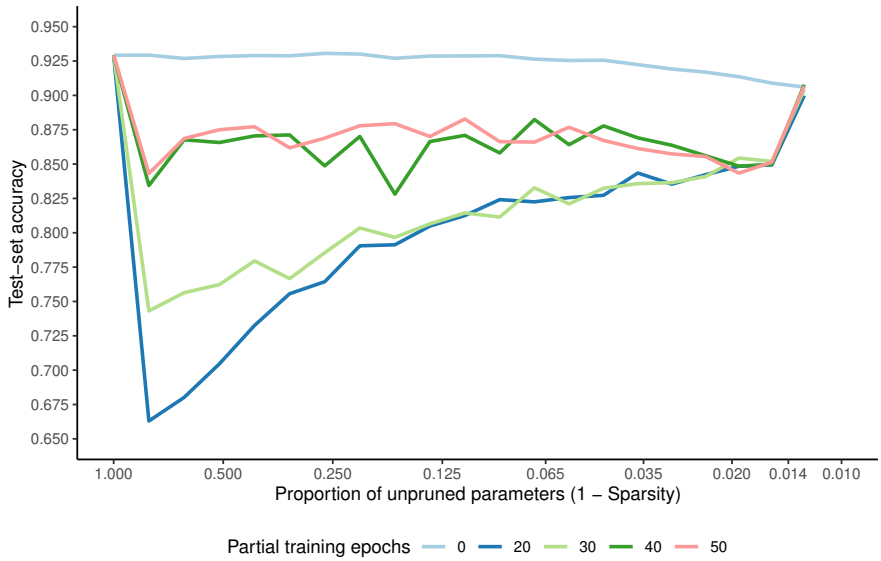


Figure 3.12: Test set accuracy during the various pruning iterations for experiment (i): AIMP + WR pruned with various values of partial training epochs τ. $\tau = 0$—i.e., all retraining iterations operated with 160 epochs—refers to the model trained with UIMP + WR. x axis in log scale for the sake of visualization.

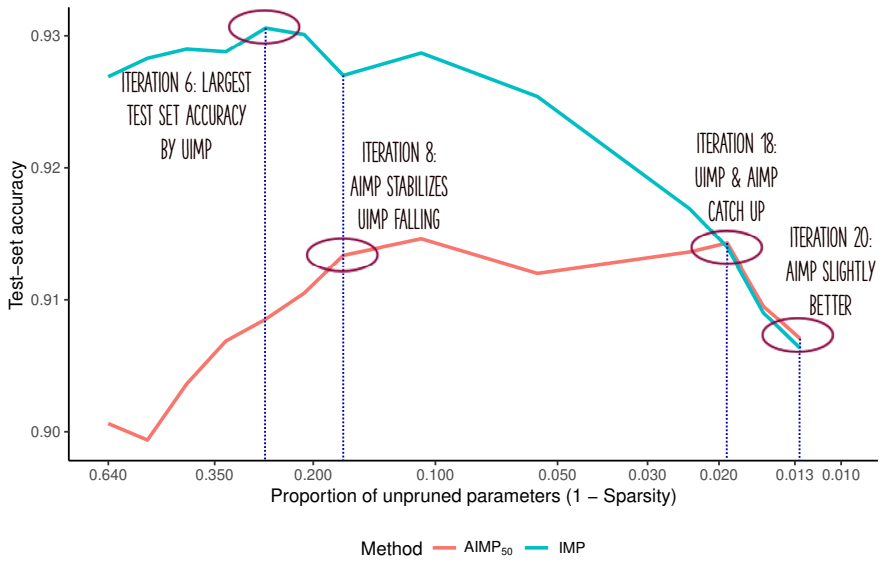


Figure 3.13: Test set accuracy for various levels of sparsity for experiment (iv): AIMP + WR pruned with $t_{\text{prune}} \in \{3, \ldots, 20\}$. x axis in log scale for the sake of visualization.

instability, i.e., a configuration which would lead to dramatic differences in pruned parameters if pruning would be applied after these epochs. Such condition did not occur in our case.

- The L2 and L∞ norms of the top-1000 parameters in magnitude. We chose the top parameters as we assumed those to be the most resistant to pruning. This criterion was inspired from Frankle et al. [35]. Similarly to the potential pruning mask, we wanted to investigate the existence of dynamics in the parameters that would suggest $20, \ldots, 50$ to be possible values for τ. Also in this case, no clear indications emerged.

3.2.7 Conclusions, limitations, contextualization

In [146] we showed that UIMP, an effective-yet-heavyweight pruning technique, could be sped up significantly by stopping early (at an epoch τ) the retraining in all but the last pruning iteration. We dubbed the technique proposed by us Accelerated Iterative Magnitude Pruning, AIMP. Albeit on a single combination of CNN architecture and datasets, we empirically showed, in different settings, the pruned CNNs produced by AIMP to be competitive w.r.t. the counterparts produced by UIMP when the final sparsity of the CNN was very high (indicatively $> 98\%$ in our case). Despite the impressive results, after an analysis on the pruning masks and norm of the parameters, we were unable to find clear indications suggesting what would be an optimal value of τ. The selection of this hyperparameter would need to be operated empirically before training and could not be chosen *dynamically* during each retraining step.

Additionally, the research was limited as it lacked further investigations on other ANN architectures, datasets (e.g., ImageNet), and task domains (e.g., object detection, NLP, *etc.*). In addition, we experimented only using unstructured pruning, while our proposed paradigm could easily be adapted to structured pruning.

To conclude, our work fits into a line of other researches [137, 138], roughly from the same period as ours, showing possible ways to accelerate UIMP, although (a) AIMP performs well at high sparsity rates,

while these other methods work best at 50-70% sparsity, and (b) they are not close to challenging UIMP + LRR.

4 Applications

Next, we are going to present two works in which pruning was used as a tool to reach other goals. The first work, presented in Section 4.1, involves the utilization of both pruning and quantization in order to implement an YOLO-based object detection model for the task of face mask detection. In the second work, presented in Section 4.2, we applied pruning on ANNs acting as controllers of simulated soft robots in order to investigate possible differences in performance of these soft robots in walking-related tasks on a variety of terrains.

4.1 YOLO-based face mask detection on low-end devices using pruning and quantization

The COVID pandemic, started at the beginning of 2020, has impacted the lives of many individuals throughout the world: it has forced lockdowns and, more generally, has imposed governments and citizens to revisit a number of undervalued sanitary practices and introduce new ones. For instance, the use of face masks in the western world was strictly limited to, e.g., specific hospital departments; after the breakout of COVID, most nations introduced face mask mandates imposing the citizens to wear face masks in closed spaces, or, in some cases, even in open air. This has caused the necessity, for the personnel of shops, supermarkets, libraries, schools, *etc.* to act as controllers for checking that customers were wearing face masks correctly. This has prompted many researchers to develop automated systems for carrying out this task, hopefully relieving the need for human controllers. In the last 2 years, many [1, 15, 26, 58, 60, 62, 67, 96, 107, 141] works have been dedicated to performing object detection for recognition and localization of face masks using DL-based approaches. Rarely [62], though, the

focus of these works has been on efficient implementations on lower-end devices, which are usually the hardware which we can find in devices for video surveillance: in fact, while models like YOLOv4 require high-end hardware, such as CUDA-capable GPUs, to operate inference in real-time, having a model of this size and specifications run many hours a day (sometimes even 24/7) can lead to unfeasible levels of energy consumption. For this reason, we decided to concentrate on *scaling down* an object detector based on YOLOv4 in order to run on a single-board computer, a Raspberry Pi 4, which we deemed sufficiently *small* and *low-energy* for our goal. Interestingly, this is the same device that Kong et al. [62], authors of the only work with goals aligned to ours, used in their implementation. The object detector we proposed was evaluated according to two metrics: Average Precision (AP), which quantifies detection accuracy, and Frames-per-Second (FPS), which measures speed of inference. These two are competing goals, as, usually, increasing AP requires more complex models, which, on the other hand, decrease FPS, and *vice-versa*. Our findings revealed that pruning and quantization increased the FPS by almost 2 times w.r.t. the original, unpruned model, while keeping a respectable AP.

Our work was published: Liberatori et al. [74].

4.1.1 Details on the YOLOv4 implementation

For this work, we resorted to building upon a variant of YOLOv4-tiny, originally developed by Jiang et al. [59]. We preferred YOLO to other object detection architectures due to a number of factors, such as (a) it, being a one-shot detector, is faster than other two-shot variants and it targets speed over accuracy, disregarding the exact localization of the bounding boxes it outputs, and (b) popularity: YOLO has a large number of implementations freely available on-line, especially the *tiny* versions [9], which are optimized for lower-end GPUs. The version of YOLOv4-tiny we used has the following characteristics:

- It makes extensive use of bottleneck residual blocks (see Section 2.2.4), specifically a variant called ResNet-D block [47], which is optimized for image classification on ImageNet. YOLOv4, on

the other hand, uses another block called CSPBlock [131], whose inefficiency w.r.t. residual blocks was already known [129]. A CSPBlock is nonetheless used before the DHs. A schematic depiction of it is presented in Figure 4.2.

- It enriches the ResNet-D blocks with channel-wise and spatial attention mechanisms (called CBAM) [133], which are able to weight the channels and spatial coordinates respectively where the most important features for the task at hand are located. Assuming an activation $A \in \mathbb{R}^{c \times h \times w}$, the channel-wise attention introduces a learnable vector $U_{\text{channel}} \in \mathbb{R}^c$ which weights the most important channels; similarly, spatial attention introduces a learnable matrix $U_{\text{spatial}} \in \mathbb{R}^{h \times w}$. By replicating $U_{\text{channel}} \in \mathbb{R}^c$ across the spatial dimension, we get a tensor $U'_{\text{channel}} \in \mathbb{R}^{c \times h \times w}$, while replicating U_{spatial} across the channel dimension, we get a tensor $U'_{\text{spatial}} \in \mathbb{R}^{c \times h \times w}$. First, the channel-wise attention is applied to the incoming activation: $A' = A \odot U'_{\text{channel}}$. Subsequently, the spatial attention is applied: $A'' = A' \odot U'_{\text{spatial}}$. A'' is the output of CBAM. A schematic depiction of CBAM is shown in Figure 4.1.

- It replaces the Mish activation function [84] used in YOLOv4 with the more efficient Leaky ReLU:

$$\text{Mish}(x) = x \cdot \tanh(\log(1 + e^x))$$

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}, \alpha \in [0, 1).$$

  A plot of the two functions is visible in Figure 4.3.

- It uses two DHs (while YOLOv4 uses at least three) to speed up inference. This, of course, leads to a smaller accuracy, as an additional DH could help in recognizing objects at a finer or coarser resolution, depending on how the DH itself is implemented. In addition, the path leading to the DHs (*neck*) has been greatly simplified, as the one implemented in YOLOv4 has more interconnections between the paths leading to the heads.

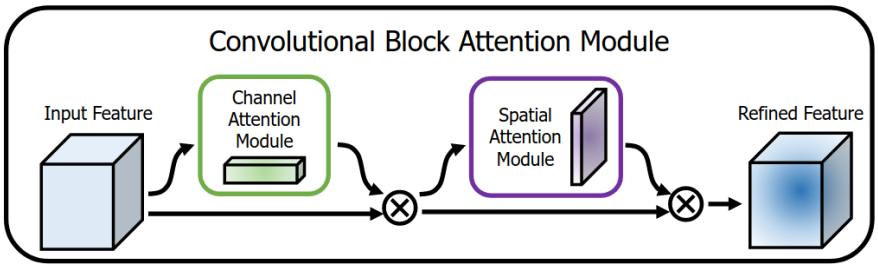


Figure 4.1: A schematic depiction of the CBAM mechanism. The incoming activation ("input feature") undergoes channel-wise attention and spatial attention for the *highlighting* of the most important channels and spatial coordinates in terms of useful features for the task at hand. Image from [133].

The architecture proposed by Jiang et al. [59] is schematized in Figure 4.4.

4.1.2 The dataset

For our experiments, we used a publicly available dataset called "Mask-Detection-Dataset" [14]. The dataset is composed of 6766 color images depicting a variable number of people or human faces at various scales, in a variety of orientations, sizes, and locations (both open and close spaces). All images are supplied with a text file containing the labels, which specify (a) the coordinate of the bounding box surrounding the face, and (b) the category (face mask worn/not worn). The dataset was already arranged in two splits: (a) a trainset of 5448 images, and (b) a test set of 1318 images. A selection of images from the trainset is presented in Figure 4.5.

Despite the existence of a number of datasets for face mask detection, we opted to use this dataset for factors such as (a) free availability on-line, (b) variability (i.e., people in different poses, variable number of people in the images, *etc.*), (c) quality of the labels (i.e., large number of incorrectly classified instances), and (d) size of the dataset, as, with the tools at our disposals, it would have been difficult to train the model

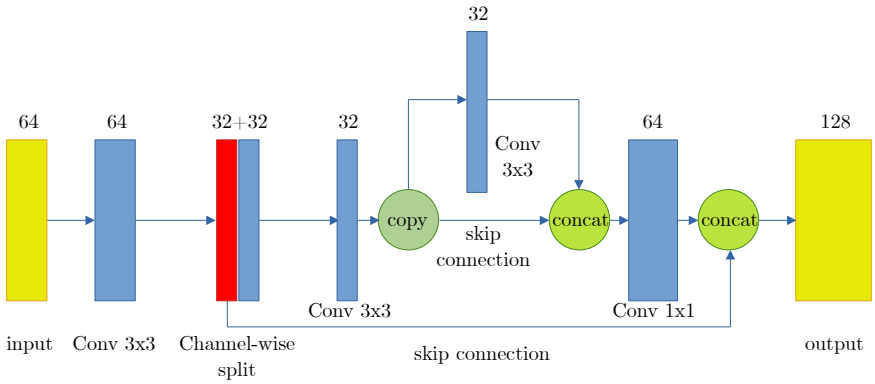


Figure 4.2: Schematization of the CPSBlock. It can be seen as an extension of a residual block of a ResNet—see Section 2.2.4. The input, having a given channel dimension c (in the image, 64), passes a convolutional layer with a kernel size of 3×3 ("Conv 3×3") and is subsequently *split* alongside the channel dimension. The first $c/2$ channels are skipping the subsequent convolutional layers (essentially forming a skip connection), while the last $c/2$ channels undergo another Conv 3×3, another Conv 3×3 with a skip connection, and, finally, another Conv 1×1, whose output is concatenated to the previous $c/2$ channels that have skipped the convolutional layers. All convolutional layers leave untouched the spatial dimension of their input. BNs and activation functions are omitted for simplicity. Bochkovskiy et al. [9] suggest using the Mish activation function [84]. Notice that, conversely to ResNets, the skip connection does not end with a sum of the two paths, but with a concatenation ("concat"), thus increasing the channel dimension, instead of leaving it constant.

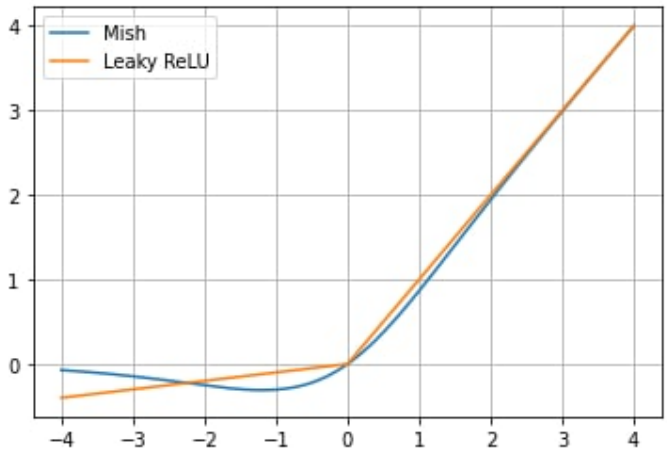


Figure 4.3: Plot of the Mish (in blue) and Leaky ReLU (in orange) activation functions. Mish (a) is differentiable everywhere, and (b) has a phase transition around -1 (i.e., is not monotonic) which has been shown to be beneficial to training. Despite all of that, Leaky ReLU is far more time-efficient, and thus is to be preferred when targeting implementations on low-end devices.

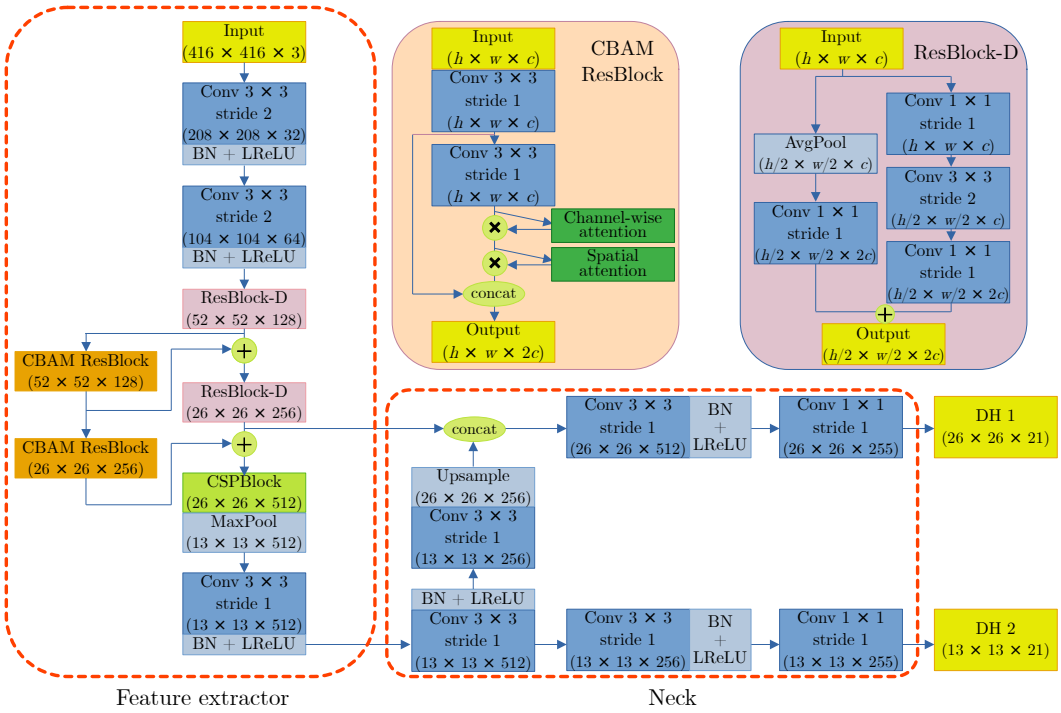


Figure 4.4: Diagram of the YOLOv4tiny variant proposed by Jiang et al. [59], with detail of the CBAM ResBlock and ResBlock-D. The FE has a combination of convolutional layers ("Conv $k \times k$", where k is the kernel size), ResBlock-D's and CBAM ResBlocks, which implement channel-wise and spatial attention, and a CSPBlock. "LReLU" refers to the Leaky ReLU activation function. "Upsample" operates an upsampling via a *transpose convolution* [31] with learnable parameters. The channel dimension of the output of the DH is 21, which is 3 (the number of anchor boxes) times 7, this number being 4 (the coordinates of a bounding box) plus 2 (the number of classes, i.e., mask worn or not worn) plus 1 (o, the confidence in the prediction). In the CBAM ResBlock and ResBlock-D diagrams, all convolutional layers apply BN and Leaky ReLU (omitted for simplicity).

Figure 4.5: Selection of pictures from the trainset. The images, collected from diverse sources, depict various people, with or without a face mask, in different situations. The original images come in different sizes: in the figure, they are resized to a common dimension for the purpose of representation.

on a large-scale dataset. Kumar et al. [67] presents a list of datasets for face mask detection, indicating how all of them have some issues (i.e., they are missing the labels); nonetheless, their dataset lacks quality as it is already pre-augmented (i.e., the images are not provided in their original form, but with geometric/color transformations applied).

Data preprocessing and augmentation

Despite the dataset being *per se* already suitable for training an object detector, we decided to add a preprocessing step and an augmentation routine.

The preprocessing step is necessary as the model will have to be deployed on a Raspberry Pi with a low-quality, low-resolution camera: thus, we wish to artificially *degrade* the quality of the images to reproduce the low-fi scenario at which the model would be deployed. The preprocessing is carried out with by applying the following transformations:

1. Resizing all the images to have a height and width of 416 pixels. This is also needed for performance issues, as otherwise it would be difficult to evaluate multiple images efficiently: if they all share the same size, they can be bundled up in a tensor of dimension $m \times 3 \times 416 \times 416$ in our case, m being the number of images. While doing inference on such a structure, the processor is able to take advantage of all sorts of optimizations, thus rendering this evaluation much faster than, e.g., evaluating the m images sequentially.

2. Down-scaling of a factor of 75%. Downscaling is implemented by downsampling the image with the desided factor, then upsampling it to the original size. In our case, we downsampled and upsampled using nearest neighbor interpolation.

3. Motion blur added by cross-correlating the image with a kernel of zeros, barring a "line" of 1s (e.g., the ones are positioned on a single column or row or even diagonal).

Figure 4.6: Selection of images from the trainset after the application of preprocessing + augmentation.

The augmentation step instead aims at artificially increasing the variability of the dataset by means of geometric or color transforms. In our case, the augmentation routine was composed of the following transforms:

1. Random rotation with probability 0.5 with an angle randomly selected in the interval $[-20°, 20°]$, this in order to increase the orientation of the objects that the model is tasked with detecting.

2. Horizontal flipping of the image with probability 0.5.

Figure 4.6 showcases the application of preprocessing on a selection of images from the trainset.

In our implementation, we opted for directly applying preprocessing and augmentation during training, specifically during the shuffling at the beginning of each epoch of SGD (see Section 2.2.2), instead of applying it before training. This ensures that the stochastic elements of

preprocessing and augmentation are to be re-applied each time: the model will then hopefully evaluate the same image with, e.g., a different rotation and a different flavor of motion blur. Moreover, while evaluating the model with the test set, we still applied preprocessing, still trying to reproduce the low-fidelity of the Raspberry Pi. Preprocessing, instead, is not applied when deploying the model on-the-wild, e.g., when receiving images from the Raspberry Pi's camera.

4.1.3 Training

In this section, we are going to present how the training was operated. We will detail the loss function used and explain the proper training.

Loss function

The loss function we used is the one introduced in the original YOLO implementation [103]. It is a quadripartite loss composed of three terms for enforcing a good localization of the bounding box and a good classification:

$$C = \lambda_{\text{box}} C_{\text{box}} + \lambda_{\text{no-obj}} C_{\text{no-obj}} + \lambda_{\text{obj}} C_{\text{obj}} + \lambda_{\text{class}} C_{\text{class}}. \tag{4.1}$$

The loss is calculated *after* the output of the DHs has been upscaled to match the image size, so, we are left with a grid of a fixed number of squares, some of which are formulating one or more predictions (up to 6 in our case). The four terms are designed as follows:

- C_{box} is based on the mean-square error between the coordinates of the ground truth and the predictions, and enforces a good localization of the bounding box.
- C_{obj} and $C_{\text{no-obj}}$ are based upon the Binary Cross Entropy (BCE) function and penalize, respectively, situations in which the model does not predict the presence of a generic object and, conversely, situations in which the model predicts an object which does not have a corresponding ground truth. Notice that these two losses do not penalize the model in case of misclassification (i.e., a wrong object category is predicted): they only operate with reference to

the confidence score o (also called *objectness score*) enclosed in each prediction of the DHs. Given $y \in \{0, 1\}, \hat{y} \in [0, 1]$, respectively ground truth and prediction, BCE is calculated as:

$$\text{BCE}(y, \hat{y}) = -y \log(y) - (1 - y) \log(1 - \hat{y}). \tag{4.2}$$

In our case, y is a 0/1 scalar indicating whether, in a given cell, an object is or is not present, while $\hat{y}$ refers to the objectness score. Then,

$$C_{\text{no-obj}} = \mathbf{1}_{y=0}(-\log(1 - \hat{y})).$$

This function is applied wherever a given cell has no object inside ($\mathbf{1}_{y=0}$), so the connection to Equation (4.2) is immediate. Similarly, we have

$$C_{\text{obj}} = \mathbf{1}_{y=1}(-\log(\hat{y})).$$

- Finally, C_{class} is the regular CE from Equation (2.2) comparing the categorical ground truth with the predicted category (n.b., the categories are "mask worn" or "not worn").

Notice that the equations presented above are specifications for single training examples. For extending the loss to multiple examples, like a training batch, all of the four terms need to be averaged on these instances. The coefficients $\lambda_{\text{box}}, \lambda_{\text{no-obj}}, \lambda_{\text{obj}}, \lambda_{\text{class}}$ are scalars which can be tuned to render one term more prevalent w.r.t. the others. In our implementation, we set $\lambda_{\text{box}} = 1, \lambda_{\text{no-obj}} = 5, \lambda_{\text{obj}} = 10$, and $\lambda_{\text{class}} = 1$.

Proper training

We trained the model for 100 epochs using a batch size of 32. We used the optimizer RAdam, which we already presented in Section 2.2.2, using a LR $\eta = 0.002$ (tuned upon running a grid search over multiple values) and the coefficients $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In addition, we applied L2-norm regularization with a coefficient λ of 0.005. Both the implementation and training were done on Python 3.8 with the library PyTorch version 1.8. The model was trained on a single NVidia V100 GPU made available to us by AREA Science Park Trieste.

4.1.4 Pruning

After training, we operated L1 norm-based filter pruning, presented in Section 2.5.1. As far as the strategies are concerned, we experimented with two approaches:

(a) OSP + FT: prune once and retrain for 50 epochs.

(b) IP + FT with LRR: prune, then retrain with LRR, but instead of retraining for the whole 100 epochs, retrain only for 5 epochs, repeat for 7 iterations.

We investigated the effect of different pruning rates π: in the case of (a), we experimented with $\pi = 0.5, 0.6, 0.7, 0.8$, and 0.9; for (b), instead, we tried with $\pi = 0.1$ and 0.2, obtaining, after 7 iterations, a total pruning rate of around 0.5218 and 0.7903 respectively. As for the training, also pruning was operated on Python + PyTorch.

4.1.5 Quantization

The final step of our pipeline is the application of quantization to INT8. We experimented both with DRQ and FIQ, introduced in Section 2.5.2. For applying quantization, we made use of the TFLite environment. TFLite allows for the optimization of models for low-end devices, like microcontrollers. It acts by converting a model from Python + TensorFlow to C, which is a language more suited for such devices. In order to use TFLite for our pruned models, we first needed to convert them from PyTorch to TensorFlow. We did so by means of ONNX[1], which allowed us to quickly translate all of the structures and the corresponding parameters to TensorFlow. Then, we applied quantization using TFLite, which retured a C implementation of the models.

4.1.6 Model assessment and results

Metrics for assessment

We assessed our models according to two metrics: AP, which measures the detection accuracy, and FPS, which quantifies the speed of

[1] `https://onnx.ai/`

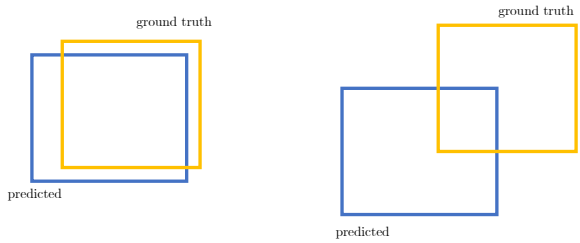
inference. The calculation of AP first requires to define a criterion according to which a predicted and a ground truth bounding boxes are defined as "matching": this metric is the *Intersection-over-Union* (IoU) or Jaccard similarity. It is a metric, originally developed for measuring the similarity of sets, but it can extend to the task of assessing quantitatively the overlap between two generic areas, such as the bounding boxes for object detection. In our case, let us denote with Area_y and $\text{Area}_{\hat{y}}$ two generic bounding boxes, the first referring to a ground truth, the second to a prediction. IoU is calculated as

$$\text{IoU}(\text{Area}_y, \text{Area}_{\hat{y}}) = \frac{\text{Area}_y \cap \text{Area}_{\hat{y}}}{\text{Area}_y \cup \text{Area}_{\hat{y}}}.$$

In the context of AP, IoU is used to determine a threshold above which two bounding boxes (one predicted, one ground truth) belonging to the same category are considered as *matching*. The string AP @ IoU = 0.5 refers to AP calculated by considering two bounding boxes whose category is the same and IoU is larger than or equal to 0.5. This is the specific value of threshold which we used in our implementation. We will subsequently refer to AP @ IoU = 0.5 simply as "AP", dropping the value of the threshold for readability purpose. This procedure is exemplified in Figure 4.7.

The next step is the computation of the confusion matrix for the match between the bounding boxes predicted and the ground truths. The confusion matrix is computed as such:

True Positive (TP) Predicted bounding box match ground truth	**False Positive (FP)** Predicted bounding box does not match ground truth
False Negative (FN) Ground truth with no corresponding prediction	**True Negative (TN)** Bounding box not predicted, no ground truth



IoU $>$ 0.5
match

IoU $<$ 0.5
no match

Figure 4.7: Usage of IoU with threshold of 0.5 to determine a match between a predicted and a ground truth bounding boxes.

The precision for the class $j \in \{1, \ldots, k\}$ is then computed as $\text{Precision}^{(j)} \doteq \frac{\text{TP}}{\text{TP+FP}}$. Finally, AP is calculated as the mean of the precision for all the categories:

$$AP = \frac{\sum_{j=1}^{k} \text{Precision}^{(j)}}{k}.$$

Notice that this formulation of AP was the one used as evaluation metric in the popular Pascal VOC challenge for object detection[2] and is still extensively used in the literature. The proper AP should consider a broader range of thresholds, as in the COCO challenge[3], which is, of course, much expensive to evaluate computationally.

Frames-per-Second (FPS), our second assessment metric, measures the speed of inference of a model. It is calculated as the time needed to obtain the prediction on one single image on a given processor (even a GPU).

[2] `http://host.robots.ox.ac.uk/pascal/VOC/`
[3] `https://cocodataset.org/#detection-eval`

Table 4.1: Number of parameters, FPS, and AP @ IoU = 0.5 for the models pruned with the various pruning schedules indicated in Section 4.1.4. "baseline" refers to the unpruned model, shown for the sake of comparison. FPS reported are all computed on the Raspberry Pi 4 after converting the models to TFLite.

model	# parameters	FPS	AP@IoU = 0.5
baseline	9.1M	0.99	0.618
`one-shot`$_{0.5}$	3.1M	1.59	0.584
`one-shot`$_{0.6}$	**2.5M**	**1.67**	**0.587**
`one-shot`$_{0.7}$	2.1M	1.80	0.489
`one-shot`$_{0.8}$	1.8M	1.93	0.446
`one-shot`$_{0.9}$	1.5M	**2.07**	0.233
`iterative`$_{0.1}$	3.0M	1.51	0.601
`iterative`$_{0.2}$	1.8M	1.93	0.511

Results

We first report the results attained for the pruned model. As indicated in Section 4.1.4, we pruned the model using SMP with various pruning rates and retraining paradigms. We will refer to as the models having undergone OSP with FT as `one-shot`$_\pi$, with π being the pruning rate applied. Similarly `iterative`$_\pi$ indicates the models pruned with iterative SMP, with π specifying the pruning rate used at each pruning iteration. The results in terms of AP and FPS are reported in Table 4.1.

We can notice the following:

- The best model in terms of FPS is, expectedly, the one with the highest pruning rate, `one-shot`$_{0.9}$, although it attains a value of AP 62% worse than the dense model.
- In terms of mediating between FPS and AP, the best model seems to be `one-shot`$_{0.6}$, which increases FPS by 69% while losing only 5% of AP.

Table 4.2: Number of parameters, model size in MB, FPS and AP @ IoU = 0.5 for "baseline" model (unpruned and unquantized) and $\texttt{one-shot}_{0.6}$ before and after quantization. FPS reported are all computed on the Raspberry Pi model 4 after converting the models to TFLite.

model	# parameters	model size (MB)	FPS	AP@IoU = 0.5
baseline	9.1M	36.5	0.99	0.618
$\texttt{one-shot}_{0.6}$	2.5M	12.3	1.67	0.587
$\texttt{one-shot}_{0.6}$ `DRQ`	2.5M	2.7	1.48	0.586
$\texttt{one-shot}_{0.6}$ `FIQ`	**2.5M**	**2.7**	**1.97**	**0.574**

- $\texttt{iterative}_{0.1}$ still records a good AP, despite not reaching FPS as good as $\texttt{one-shot}_{0.6}$.

Thus, we decided to elect $\texttt{one-shot}_{0.6}$ as the best pruned model and to proceed with the application of quantization to this model. As anticipated in Section 4.1.5, we experimented with both DRQ and FIQ. Results are shown in Table 4.2. As for the name of the model, we append `DRQ` or `FIQ` to the string to specify the type of quantization used, e.g., $\texttt{one-shot}_{0.6}$ `FIQ` refers to $\texttt{one-shot}_{0.6}$ with further application of FIQ.

From the table we can immediately notice how quantizing with FIQ seems to produce the best-performing model. The FPS almost double with respect to the baseline model and increase by 18% with respect to the pruned model; in addition, the model loses 7% of AP to the baseline, and only 2% to the pruned model. DRQ, on the other hand, is able to almost match the AP of the pruned model, but we get a decrease in FPS, probably due to the large computational overhead on the small processor of the Raspberry Pi. This overhead is caused by the calculation of the dynamic ranges and the conversion from FP32 to INT8 of the data and activations.

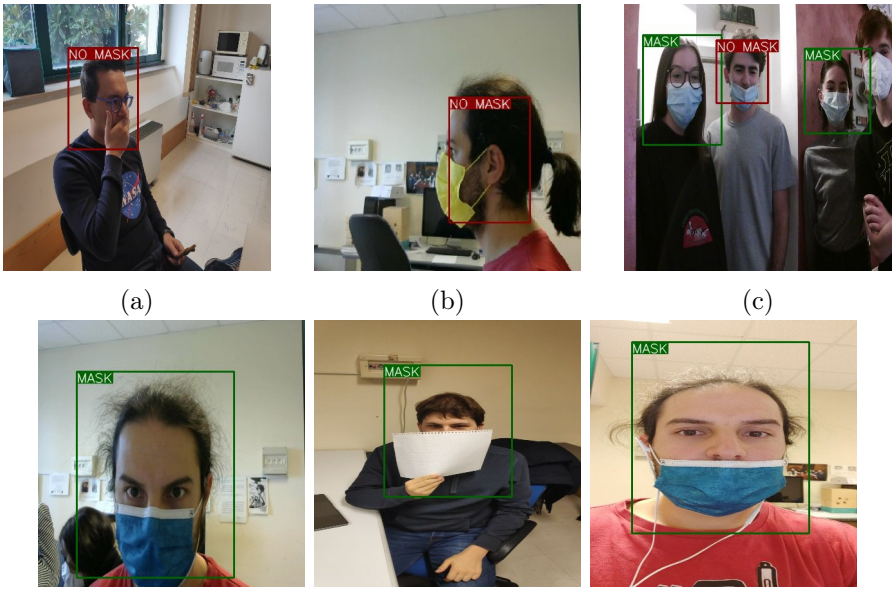
All in all, we selected $\texttt{one-shot}_{0.6}$ `FIQ` as the final model, which we deployed on the Raspberry Pi.

We have encontered problems regarding comparison with other ex-

isting works presenting face-mask detection systems, as (a) most of the existing publications did not release their code, and/or (b) the great majority of the works did not target implementations in low-end devices, as previously noted when introducing our research. Kong et al. [62] were the only ones at the time to deploy their model on a low-end devices, although they did not release their code. Despite implementing their solution onto a Raspberry Pi 4, they also used a Neural Compute Stick (NCS), an external device to augment the computational capabilities of the Raspberry Pi for matrix-matrix multiplications. We did not own an NCS, so the FPS that they reported is not at all comparable to ours. Regarding AP @ IoU = 0.5, we see that most of the other works, despite using different datasets, reported results in line with ours.

Evaluation of the model on-the-wild Finally, we evaluated $\texttt{one-shot}_{0.6}$ `FIQ` qualitatively on-the-wild, by running it along with a webcam. Figure 4.8 presents a set of six images which we deemed to summarize well the strengths and weaknesses of this model when evaluating images outside of the training/test set.

In Figure 4.8a, we show how the model is not fooled by a person covering his mouth with a hand. Conversely, in Figure 4.8e we see how covering the mouth with a sheet of paper does indeed fool the model. In Figure 4.8b we can notice that the model seems to have some difficulties in recognizing masks when people are posing in profile. Figure 4.8c showcases the functioning of the model in pictures with multiple faces. It correctly recognizes the man in the center as not wearking a mask as it is worn below its nose; on the other hand, it can't recognize the presence of the man on the right, probably due to the close vicinity to the frame of the image. In Figure 4.8f, though, we can see that the model is unable to detect that the mask is worn below the nose, and erroneously flags the man as correctly wearking the mask. Finally, in Figure 4.8d, we can see that the woman standing in the background (on the bottom-right of the man in the foreground) is not detected: YOLO has been in fact noted to have problem in detecting multiple object being close by, probably due to the fact that DHs are designed in a grid structure [56]. The issues with recognizing profile poses and masks



(a) (b) (c)

(d) (e) (f)

Figure 4.8: Output of our final ANN one some on-the-wild images, showcasing strengths and weaknesses of our proposed model.

incorrectly worn are instead probably attributable to the dataset, as few or no instances with these characteristics exist.

4.1.7 Conclusions

In [74], we developed an object detection model based on YOLOv4 for detecting whether individuals in a picture are wearing or are not wearing face masks. We wanted our model to be deployed on a low-end device as we envisoned its application akin to a video surveillance tool: in this field, in fact, often devices with limited computational capabilities are employed in order to limit the energy consumption, as having desktop computers running 24/7 might be too expensive. Starting from a lightweight implementation of YOLOv4, we trained the model, then applied SMP with different retraining schedules, and eventually applied quantization to further lighten the model and deploy it on a Raspberry Pi 4. We saw that pruning and quantization accelerated the inference phase of the model by almost 100% on the Raspberry Pi, while decreasing the AP by a meager 7%. We concluded by doing some qualitative evaluation of the detector on-the-wild, noticing some limitations on some specific poses of the subjects or when the face masks are incorrectly worn. These limitations might be tackled by using a more comprehensive dataset, encompassing also these situations.

We are currently working on an extension of this work, using YOLOv7 [130], which was released recently, and more up-to-date structured pruning techniques.

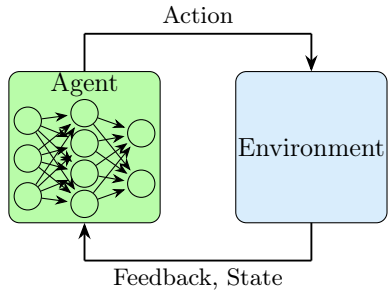


Figure 4.9: Exemplification of the functioning of the robot. The robot is an agent, controlled by a given program (in the figure, an ANN), which takes an action, interacting with the surrounding environment. The environment gives a feedback to the agent and update its state.

4.2 Application of pruning on Voxel-based Soft Robots (VSR)

In the field of robotics, the actions of robots are controlled by computer programs, called *controllers*, that lead the robots toward the completion of one or more tasks. Controllers often employ ML-based techniques (like ANNs) for processing incoming information and formulating the subsequent actions of the robot. When ANNs are used as controllers, we call them *neural controllers*.

With reference to Figure 4.9, a robot can be represented as an agent which is operating in an interactive environment. The agent, which is in a given state, interacts with the environment by taking actions. In response to these actions, the environment returns a feedback to the agent, updating its state.

In the case of an ANN controller, we can think of its input being, e.g., the readings of its *sensors* (which essentially receive feedback from the environment) and some form of embedding of its state. The ANN combines these features to produce the output, which is the action(s) that the robot needs to undertake.

In response to this, an interaction with the environment is created,

and its feedback goes back to the ANN, which in turn will produce a different set of actions, and so on and so forth in an iterative fashion. Using an ANN as a controller for a robot leads the way for a large number of investigations regarding the possible parallelisms between artificial and biological neural network, as the ANN can be essentially thought of as an artificial *brain*. It is, thus, a large interest of the community of robotics to explore whether these two kinds of neural networks are exhibiting similar properties, or whether they react similarly to given stimuli, or whether they work out similar solutions. In our specific case, we investigated the application of pruning on ANN controllers. Synaptic pruning is a process occurring normally in human and animal brains [80] and the process underlying it is quite complex, with a lot of factors determining the synapses to be pruned [49]. Pruning, in this context, is thought of to be useful in organizing the brain into efficient topologies [116] robust to perturbations and adaptable; in addition, biological neural networks exhibiting excessive or insufficient sparsity are linked with conditions such as schizophrenia [18], thus highlighting the importance of pruning in animal brains. In our work, we applied pruning to neural controllers of VSRs trained via NE in order to study whether the robots which they control exhibited similar performance on a task. Moreover, we investigated the level of generalization of said ANNs by deploying the robots in a variety of environments: in this sense, we can interpret the generalization as the *adaptability* of the neural controller to function in different environments than the one in which it learned its task. The contribution of this work was a thorough and in-detail investigation of the effects of pruning, which was, at the time, surprisingly little-studied—especially concerning biological parallelisms—in the context of neural controller trained with NE, with [39, 113] being two examples. We found that pruning, especially at low pruning rates, often has negligible effects on the performance, and that it does not impact on the generalization capability of the VSRs.

Our work was published in [87] and subsequently extended in [88].

4.2.1 VSRs

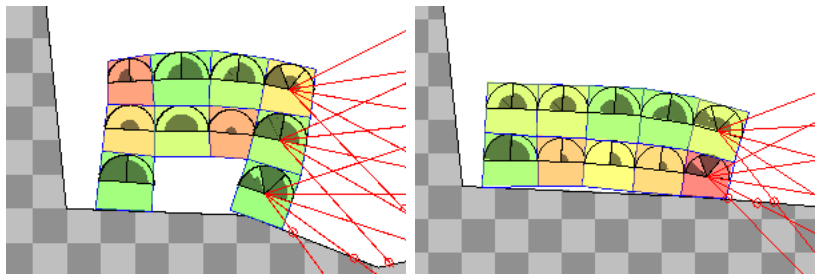
We made use of simulated VSRs, which are *modular soft robots*. *Modular* refers to the fact that the robot is composed of square blocks, called *voxels*, of the same size that can be combined to give the robot a large variety of shapes, which may be optimized for specific actions. *Soft* means that the voxels are not rigid, i.e., they can contract and expand independently. We trained these robots to *walk* on a given terrain in a given direction. The action of walking is performed by contracting/expanding the voxels such that the resulting movement of the robot produces an advancement along the designated direction of movement. Each voxel is equipped with some *sensors* recording its current area, velocity, and contact with the ground. In addition, some voxels may also have vision sensors, recording the distance from the voxel to close objects. The sensors are a way for the robot to gain feedback from the environment following the completion of a set of actions. Figure 4.10 depicts the two architectures of VSRs used in our experiments, the biped and the worm, and details the functioning of the vision sensors.

We can generally formalize the sensor readings for a generic voxel i as a vector $h^{(i)} \in \mathbb{R}^{m_i}$, where m_i, the dimensionality of these readings, is dependent upon the specific voxel, as some voxels have more sensors then others.

4.2.2 Paradigms for neural controllers

In our work, we considered two different types of controllers:

- Centralized Neural Controllers (CNCs), which are controllers composed of a single, central ANN.
- Distributed Neural Controllers (DNCs), where the controller is composed of multiple ANNs, one per voxel. The various ANNs can have all the same architecture, which puts us in a situation of *homo-distributed* DNCs; conversely, *hetero-distributed* DNCs allow for different architectures for different ANNs, thus giving more freedom of implementation to the designers. In this



(a) Biped (b) Worm

Figure 4.10: The two morphologies for the robots used in our experiments: the biped (Figure 4.10a) and the worm (Figure 4.10b). Both robots are composed of 10 voxels arranged in different fashions. The voxels in the front of the robot—i.e., on the right—are equipped with vision sensors. These sensor function by checking the presence of close objects (e.g., obstacles) along five red segments, placed at different angles, departing from the center of the voxels and directed in front of it. In the image above, the detection of an obstacle is indicated by the presence of a red dot along the segments.

section, for simplicity, DNCs will be explained in their simpler homo-distributed configuration, although we made use of both paradigms in this work.

The ANN in a CNC receives as input the concatenation of the sensor readings $h = [h^{(1)}, \ldots, h^{(D)}]$, with D being the total numbers of voxels. Similarly, the output is the actions for all the voxels. ANNs in DNCs, instead, receive the sensory input partaining only the voxel that they control. This, though, might result in a higher overhead in case of architectural search, as a large number of optimal architectures of ANNs needs to be recovered, instead of only one. In addition, to allow for intercommunication between voxels, the ANNs receive as input some signals ψ from the ν neighboring voxels. So, for the ANN controlling the i-th voxel, input $= [h^{(i)}, \psi_1^{(i)}, \ldots, \psi_\nu^{(i)}]$. The output will still contain the action a for the voxel, plus the ν signals to pass on to the neighbors (which we call $\xi_1, \ldots, \xi_\nu$), which will be used by the ANNs controlling them to determine the next actions. A schematic representation of a small DNC is depicted in Figure 4.11. In the present work, we experimented with VSRs with both CNCs and DNCs.

4.2.3 Learning to walk via NE with ES

As previously introduced, we wish to have the neural controllers of the VSRs learn the task of making the robot locomote. The learning phase was carried out on a given terrain. The learning was operated via NE, using ES as evolutionary paradigm for the ANNs. This means that (a) the ANNs underwent no architectural evolution, i.e., the architecture remained the same throughout the learning phase, and (b) the learning was operated with a population based approach: we had a population of n VSRs all performing the same task for a short amount of time t_{sim}. The VSRs performed their task, then their fitness/performance on the walking task was measured by the *average velocity* attained during the final part of the simulation $(t_{\text{assess}}, t_{\text{sim}}]$. The top 25% of the individuals according to their fitness were selected, and a new generation of $n - n/4$ individuals was generating by applying random mutations to the ANN controllers and added to the previous

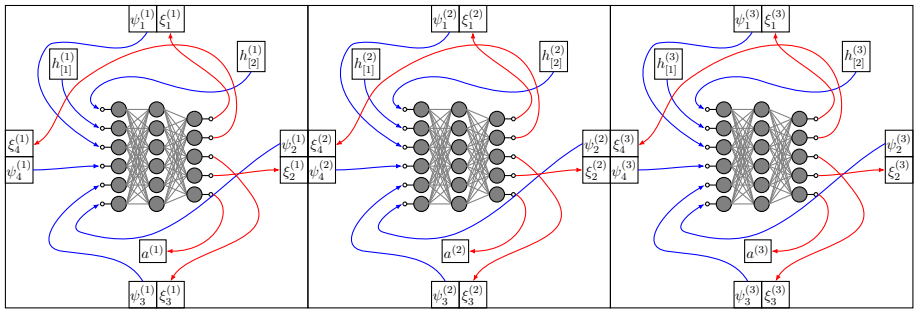


Figure 4.11: A schematic representation of a homo-distributed DNC for a 1×3 VSR with two sensors ($h^{(\cdot)}_{[1]}$ and $h^{(\cdot)}_{[2]}$) in each voxel and one communication channel per voxel side. Blue and red curved arrows represent the connection of the ANN with inputs (sensors and input communication channels $\psi^{(\cdot)}_{\cdot}$) and outputs (action $a^{(\cdot)}$ and output communication channels $\xi^{(\cdot)}_{\cdot}$), respectively. Image adapted from [88].

fittest individuals. The random mutation was performed by adding a Gaussian noise to the parameters of the ANNs. Denoting with $\Theta^{(t,i)}$ the parameters of the controller of VSR i of iteration t, we can apply mutation, generating a new generic controller for generation $t+1$ as follows:

$$\Theta^{(t+1,\cdot)} = \Theta^{(t,i)} + \mathcal{N}_p(\mathbf{0}_p, \mathrm{diag}(\varsigma)),$$

where $\mathbf{0}_p$ indicates the zero-vector of size p; ς is a vector of size p, $\mathrm{diag}(\varsigma)$ constructs a diagonal matrix of size $p \times p$ from the vector ς, and $\mathcal{N}_p(\mathbf{0}_p, \mathrm{diag}(\varsigma)$ is the 0-mean multivariate Gaussian distribution in p dimensions with covariance matrix $\mathrm{diag}(\varsigma)$. The parameters at the first generation $\Theta^{(1,\cdot)}$ were initialized by sampling from a uniform distribution $\mathcal{U}_p(-1, 1)$.

4.2.4 Pruning

We chose to experiment with applying UMP (Algorithm 1), in both its data-free and data-driven implementations, with four pruning rates: 0.125, 0.25, 0.5, and 0.75. Notice that, in the case of DNCs, the ANNs

were pruned separately, thus resulting in different sparsity patterns in the various ANNs controlling the same VSR. Recalling that each generation of controllers was evolved for an amount of time t_{sim} and assessed after an instant $t_{\text{assess}} < t_{\text{sim}}$, we decided to introduce pruning during this evolution phase *before* starting the assessment phase: $t_{\text{prune}} < t_{\text{assess}} < t_{\text{sim}}$. Drawing parallelisms with the retraining strategies presented in Section 2.5.1, our procedure would conform to OSP with FT: the training of the dense ANN corresponds to the evolution in the time interval $[0, t_{\text{prune}}]$, while the retraining of the pruned ANN corresponds to $(t_{\text{prune}}, t_{\text{sim}}]$.

In order to compare the final fitness of the VSRs with pruned ANNs with the dense ones, we also trained a population of controllers without applying pruning. In addition, for the sake of comparison, we also evolved some VSRs whose ANNs were pruned randomly with the same pruning rates of 0.125, 0.25, 0.5, and 0.75.

4.2.5 Architectural search for the neural controllers

Since the variant of ES we employed does not allow for architectural evolution of the ANNs, we repeated the experiments by considering different ANN architectures. We experimented by designing MLPs with 0, 1, and 2 hidden layers with a number of neurons for each hidden layer equal to the number of input layer. All the different architectures obtained were used, along with the different specifications (VSR morphology, controller paradigm, pruning technique, *etc.*), and their results assessed in the next section.

4.2.6 Results

As previously stated, we assessed the performance of the VSRs on their average velocity on the latter part of the simulation $(t_{\text{assess}}, t_{\text{sim}}]$.

Pruned vs. dense performance

The first comparison we operated was to measure the differences in performance between the pruned and dense ANNs, taking into consideration the differences in ANN topologies (i.e., 0, 1, and 2 hidden layers),

neural controllers paradigms (i.e., CNC, Homodistributed DNC, Heterodistributed DNC), pruning paradigms (data-free, data-driven UMP, random) and rates, and VSR morphologies (i.e., biped or worm). Figure 4.12 presents the results for different ANN topologies, neural controllers paradigms, pruning rates π, and pruning paradigms. Notice that the performance of the VSRs with dense controllers are located at $\pi = 0$. We can observe the following:

- Pruning randomly leads to substantially lower performances.
- Pruning with UMP has far better results than random pruning: in some cases, the performance drops singificantly (e.g., heterodistributed DNCs with 1 and 2 hidden layers), in other cases it improves w.r.t. the dense ANN, like in the case of CNC with no hidden layer. Anyway, by applying a Mann-Whitney U-Test [82], we obtained a quantitative confirmation that around 50% of the possible combinations of controller paradigm, ANN topology, pruning paradigm, and pruning rate, the performance of the VSR with pruned controller is not significantly different from the one with dense controller.
- Pruning with UMP and low pruning rates (≤ 0.25) leads to no substantial difference in performance.
- There seems to be no particular difference between data-free and data-driven UMP.
- All in all, it would seem that the performance of the VSRs with heterodistributed DNC are less robust to pruning, as, generally, the performance tends to decay linearly with the increase in the pruning rate. This phenomenon is much less notable with the homodistributed DNCs or the CNCs.

Next, Figure 4.13 presents the performance of the VSRs with CNC controller for different VSR morphologies, ANN architecture and pruning rate. We can observe a similar behavior w.r.t. to what observed in Figure 4.12: there does not seem to be a significant difference between the two morphologies (biped vs. worm).

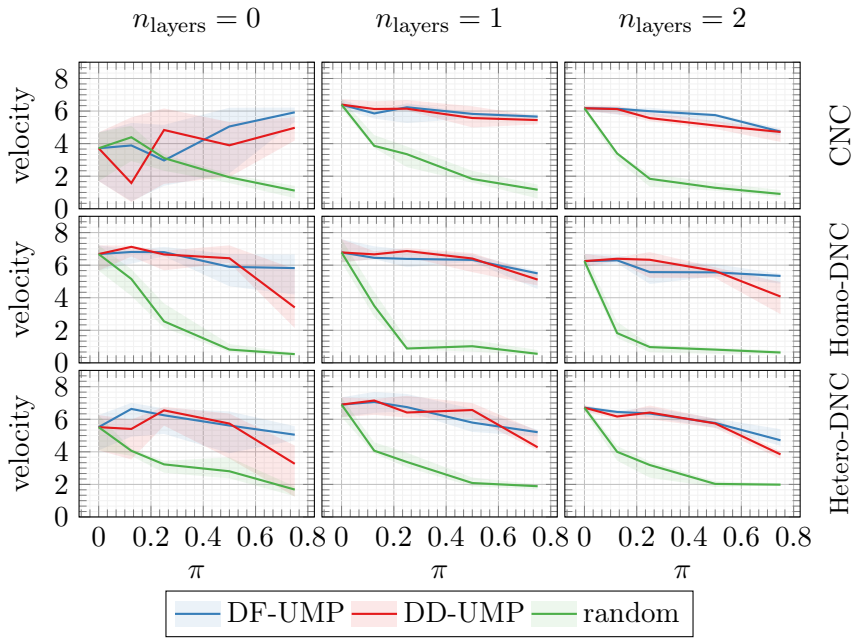


Figure 4.12: Fitness velocity (median with error bands determined by lower and upper quartiles across 10 repetitions) vs. pruning rate π, for different pruning criteria (color), controller paradigms (plot row), and ANN topologies (plot column). $\pi = 0$ corresponds to the ANNs without pruning applied. Abbreviations and acronyms: DF-UMP=data-free UMP, DD-UMP=data-driven UMP, n_{layers}=number of hidden layers, Hetero-DNC=Hetero-distributed DNC, Homo-DNC=Homo-distributed DNC. Image adapted from Nadizar et al. [88].

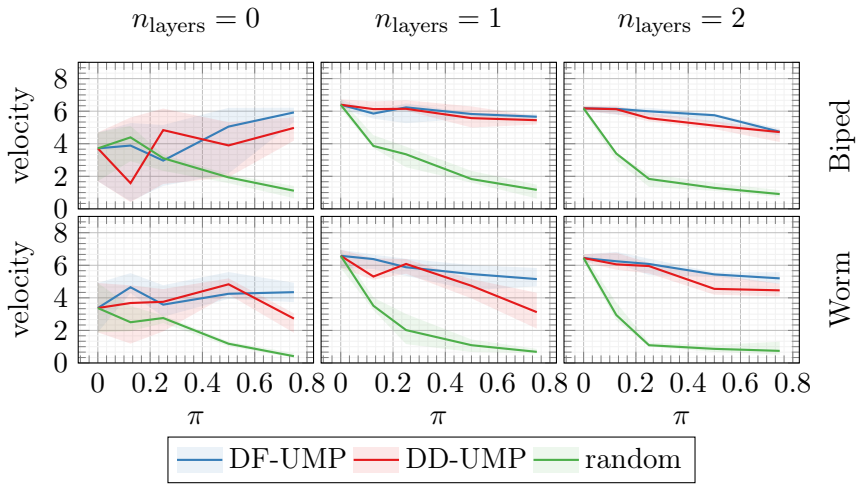


Figure 4.13: Fitness velocity (median with error bands determined by lower and upper quartiles across 10 repetitions) vs. pruning rate π, for CNC controllers evolved with different pruning criteria (color), VSR morphologies (plot row), and ANN topologies (plot column). $\pi = 0$ corresponds to the ANNs without pruning applied. Abbreviations and acronyms: DF-UMP=data-free UMP, DD-UMP=data-driven UMP, n_{layers}=number of hidden layers. Image adapted from Nadizar et al. [88].

All in all, these results tend to parallel what noted in the literature regarding pruning on ANNs trained with SGD, from early works such as [70], to more recent ones [79, 105]: pruning ANNs with UMP *can* lead to a sparse model which performs better than the dense one, but this is not always the case—the higher the pruning rate, the more difficult it is to get well-performing models.

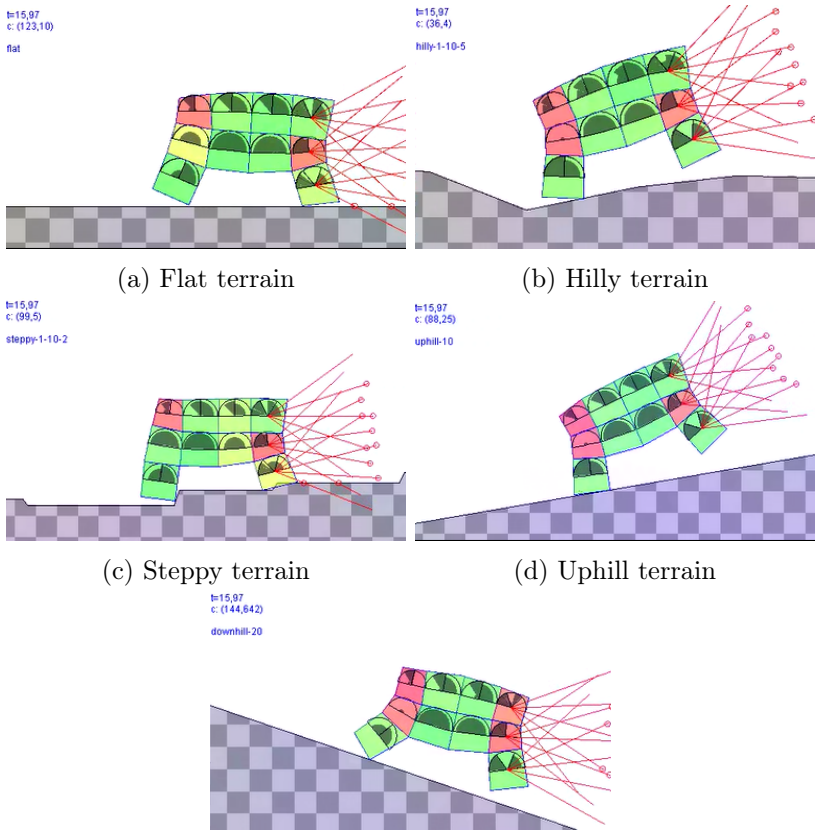
Adaptability

The second aspect we investigated is the generalization of the pruned ANN measured in terms of adaptability to carrying out the locomotion in a different terrain than the one the VSRs (and their ancestors) had experimented during their evolution. We experimented with a various set of terrains, all presented in Figure 4.14. In all cases besides the flat terrain (Figure 4.14a), the terrains have a variety of combination of hyperparameters, for incrementing the variability of the possible configurations: taking this into consideration, we tested the VSRs on 17 total different terrains. Note that, during this phase, no additional evolution was performed, i.e., the simulation was performed without modifying the parameters of the controllers.

The results are presented in Figure 4.15. In all cases, it seems that VSRs with ANNs moderately pruned ($\pi \leq 0.25$) tend to perform on-par or, in some cases (e.g., CNC with worm morphology and 0 and 2 hidden layers in the ANN), even better than the dense counterpart. These results tend to contradict partially what observed by Gerum et al. [39]: they evolved neural controllers for navigating through a maze, and observed that random pruning was improving their adaptability to perform well on larger mazes, although their result might be explained by a high level of overfitting to the specific maze by the dense controllers, which seems not to be present at a high degree in our experiments.

4.2.7 Conclusions

In [87] and [88], we experimented with applying UMP, in both the data-free and data-dependent specifications, to MLPs being part of neural



(a) Flat terrain

(b) Hilly terrain

(c) Steppy terrain

(d) Uphill terrain

(e) Downhill terrain

Figure 4.14: Different types of terrain for measuring the generalization/adaptability of VSRs with pruned controllers.

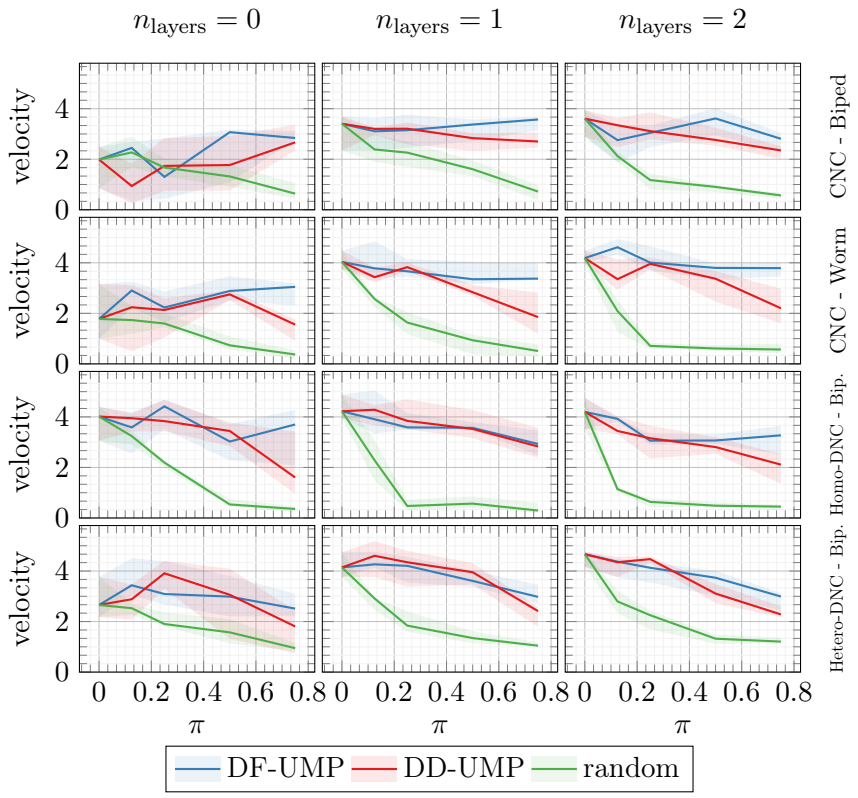


Figure 4.15: Median and error bands determined by lower quartiles and upper quartiles of re-assessment velocity vs. pruning rate π averaged across re-assessment terrains for different pruning criteria, VSR controller paradigms, VSR morphologies, and ANN topologies. Abbreviations and acronyms: DF-UMP=data-free UMP, DD-UMP=data-driven UMP, n_{layers}=number of hidden layers, Hetero-DNC=Hetero-distributed DNC, Homo-DNC=Homo-distributed DNC, Bip.=Biped.

controllers implemented in simulated VSRs. The models learned to make their robots locomote as fast as possible via NE, and were pruned during learning, thus simulating a NE-equivalent of the paradigm "prune + FT". We experimented with different MLP architectures (controlling the number of hidden layers), morphologies of the VSRs (biped and worm), controller paradigm (CNC, homodistributed and heterodistributed DNC), and pruning rates (0.125, 0.25, 0.5, and 0.75). Moreover, we evolved some controllers *without* pruning as a point of reference for comparing the performance of the pruned models, while other controllers were pruned randomly as an additional mean of comparison. The VSRs were assessed according to the average velocity attained in the final part of the simulation. The results confirmed what already observed in the literature for ANNs trained with SGD, namely, the ANNs pruned using UMP with small pruning rates often perform on-par or better than the dense counterparts, while increasing the pruning rate can incur into higher chances of degradation in the performance, especially when retraining with FT. Random pruning did seem to incur in worse velocity even at small pruning rates, which led us to discard it as an effective pruning methodology, in line with the literature for SGD [33].

In addition to this assessment, we also experimented with evaluating the velocity of the VSRs with dense and pruned controllers on terrains different to the ones they (and their ancestors) were evolved on, in order to test for their adaptability/generalization capability. The results confirmed that, also in these cases, the pruned ANNs lead the VSRs to perform similarly than their unpruned counterpart, paralleling the phenomenon that pruning leads to better generalization while training with SGD [44]. For what concerns random pruning, we observed a result contradicting another research [39], where they observed that evolved neural controller were robust to random pruning and that led to a better generalization capability, while we observed otherwise (see Figure 4.15) in our experiments.

5 Thesis conclusions

In the present manuscript, we showcased various analyses concerning pruning applied to Artificial Neural Networks (ANNs).

Pruning applied to ANNs, despite being the topic of studies in the late '80s and early '90s [70, 104], has only surged as a widespread research topic in the Deep Learning community during the 2010s, in what has been called "The New Spring of Artificial Intelligence" [12]. The motivation behind pruning is mainly related to the reduction in size of the ANNs, with the possibility of decreasing the time and resources required to train and run the models, especially when the goal is to deploy them in lower-end devices.

We subdivided our works into two logical macro-areas, namely (a) experimental investigations, where we studied specific features of pruning, and (b) applications of pruning, where this technique was used as a tool toward analyzing other topics.

The first experimental investigation was a study on the similarity between pruned and unpruned latent representation of hidden layers of Convolutional Neural Network (CNNs). Motivated by the prospect of gaining insights on the inner dynamics of pruning, we proceeded to train several CNNs on vision datasets, then prune them using Unstructured Iterative Magnitude Pruning (UIMP). We then proceeded to compare the representations learned by the pruned CNNs with the ones of the unpruned counterparts. We obtained mixed results, with some metrics indicating some possible trends, while others showing little to none. We concluded that the only trend that was observable in all cases was that pruning did in fact cause a "separation" from the representations learned by the unpruned CNNs; despite this, the pruned models were still able to perform effectively, in some cases even better, than their unpruned counterparts.

We then analyzed another aspect of pruning, namely the time effectiveness of UIMP: in fact, the application of UIMP requires sev-

eral retraining steps, which can result in an overall scheme orders of magnitude more expensive—in a computational sense—than the simple training of a dense ANN. We showed that, in various scenarios, these retraining steps could be rendered much shorter using a simple heuristic, without causing substantial loss in accuracy when the pruned ANN is pruned at *extreme* levels (i.e., with final sparsity $\geq 98\%$).

The first application was a practical work concerning the implementation, in a low-end device, of a CNN-based model for the detection of face masks in frames. This work was motivated by the recent COVID pandemic, which sparked the need for automatic systems to, for instance, check the compliance of face masks mandate in closed spaces. Since the original model we employed was too large to be run in real-time on the designated device, we first applied pruning, than quantization, doubling the speed at which the model produced the outputs, with a minimal loss in precision.

Finally, we explored the application of pruning in the context of simulated *soft robotics* controlled by ANNs. We applied Magnitude Pruning to these ANNs at various intensities and on different morphologies of robots and architectures of controllers. The robots, which were evolved to *locomote*, were then evaluated on this same task on a variety of terrains, some unobserved during their learning phase. We concluded that pruning generally showed small or no adverse effects, even when the robots are deployed on unseen terrains, hence, they have an adaptability comparable to (or, in some cases, even higher than) their *denser* counterparts.

All-in-all, recent developments in pruning have shown that there is still room for improvements, especially considering time-efficiency of the existing approaches: techniques such as *incremental pruning* (i.e., pruning during training) and *sparse-to-sparse training* (i.e, apply pruning to a randomly-initialized) have quickly caught up with iterative pruning in terms of accuracy [78], although, at high sparsity, advancements are still possible. Additionally, the regrowth of pruned parameters (*neuroregeneration*) has been explored in biologically-aware implementations [78, 118]. Another challenge for researchers is designing approaches to tackle other downfalls of existing pruning techniques: fairness [136], adversarial robustness [57, 71, 73], calibration [6], and

others. In addition, pruning (especially structured techniques) needs to be adapted to novel ANN architectures: for instance, the introduction of transformers [126] and their *vision* counterparts [30] did not allow for the application of neuron or filter pruning [72] due to the different operations (attention vs. cross-correlation) carried out during the inference. Techniques such as *attention head pruning* [48, 127, 140] have been specifically designed for transformers. In conclusion, research on pruning is still valuable to the Deep Learning: applied work is always good for showing possible approaches to other practitioners, while theoretical or experimental studies studies show that there is still room for improvements, especially when developing techniques which are aware of the specific limitations and characteristics of the individual architectures of ANN they are applied to.

Bibliography

[1] Sahand Abbasi, Haniyeh Abdi, and Ali Ahmadi. A face-mask detection approach based on yolo applied for a new collected dataset. In *2021 26th International Computer Conference, Computer Society of Iran (CSICC)*, pages 1–6, 2021. doi: 10.1109/CSICC52343.2021.9420599.

[2] Pranav Adarsh, Pratibha Rathi, and Manoj Kumar. Yolo v3-tiny: Object detection and recognition using one stage improved model. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 687–694. IEEE, 2020.

[3] Alessio Ansuini, Alessandro Laio, Jakob H Macke, and Davide Zoccolan. Intrinsic dimension of data representations in deep neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[4] Alessio Ansuini, Eric Medvet, Felice Andrea Pellegrino, and Marco Zullich. Investigating similarity metrics for convolutional neural networks in the case of unstructured pruning. In *International Conference on Pattern Recognition Applications and Methods*, pages 87–111. Springer, 2020.

[5] Alessio Ansuini, Eric Medvet, Felice Andrea Pellegrino, and Marco Zullich. On the similarity between hidden layers of pruned and unpruned convolutional neural networks. In *ICPRAM*, pages 52–59, 2020.

[6] Morgane Ayle, Bertrand Charpentier, John Rachwan, Daniel Zügner, Simon Geisler, and Stephan Günnemann. On the robustness and anomaly detection of sparse neural networks. *arXiv preprint arXiv:2207.04227*, 2022.

[7] Serguei Barannikov, Ilya Trofimov, Nikita Balabin, and Evgeny Burnaev. Representation topology divergence: A method for comparing neural network representations. *arXiv preprint arXiv:2201.00058*, 2021.

[8] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding batch normalization. *Advances in neural information processing systems*, 31, 2018.

[9] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[10] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] Jacques Bughin and Eric Hazan. The new spring of artificial intelligence: A few early economies. *McKinsey Global Institute*, 2017. URL `https://www.mckinsey.com/mgi/overview/in-the-news/the-new-spring-of-artificial-intelligence-a-few-early-economics`.

[13] Donald Bures. An extension of kakutani's theorem on infinite product measures to the tensor product of semifinite w*-algebras. *Transactions of the American Mathematical Society*, 135:199–212, 1969.

[14] Nageen Chand. Mask-detection-dataset, 2020. URL `https://github.com/archie9211/Mask-Detection-Dataset`.

[15] Amit Chavda, Jason Dsouza, Sumeet Badgujar, and Ankit Damani. Multi-stage cnn architecture for face mask detection.

In *2021 6th International Conference for Convergence in Technology (i2ct)*, pages 1–8. IEEE, 2021.

[16] An Mei Chen, Haw-minn Lu, and Robert Hecht-Nielsen. On the geometry of feedforward neural network error surfaces. *Neural Computation*, 5(6):910–927, 1993. doi: 10.1162/neco.1993.5.6.910.

[17] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53(7): 5113–5155, 2020.

[18] Enrico Cocchi, Antonio Drago, and Alessandro Serretti. Hippocampal pruning as a new theory of schizophrenia etiopathogenesis. *Molecular neurobiology*, 53(3):2065–2081, 2016.

[19] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[20] Marius Cornea. Ieee 754-2008 decimal floating-point for intel® architecture processors. In *2009 19th IEEE Symposium on Computer Arithmetic*, pages 225–228. IEEE, 2009.

[21] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[22] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Algorithms for learning kernels based on centered alignment. *Journal of Machine Learning Research*, 13(Mar):795–828, 2012.

[23] Nello Cristianini, John Shawe-Taylor, Andre Elisseeff, and Jaz S Kandola. On kernel-target alignment. In *Advances in neural information processing systems*, pages 367–373, 2002.

[24] Tianyu Cui, Yogesh Kumar, Pekka Marttinen, and Samuel Kaski. Deconfounded representation similarity for comparison of neural networks. *arXiv preprint arXiv:2202.00095*, 2022.

[25] MohammadReza Davari, Stefan Horoi, Amine Natik, Guillaume Lajoie, Guy Wolf, and Eugene Belilovsky. Reliability of cka as a similarity measure in deep learning. *arXiv preprint arXiv:2210.16156*, 2022.

[26] Sheshang Degadwala, Dhairya Vyas, Utsho Chakraborty, Abu Raihan Dider, and Haimanti Biswas. Yolo-v4 deep learning model for medical face mask detection. In *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*, pages 209–213, 2021. doi: 10.1109/ICAIS50930.2021.9395857.

[27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[28] Frances Ding, Jean-Stanislas Denain, and Jacob Steinhardt. Grounding representation similarity with statistical testing. *arXiv preprint arXiv:2108.01661*, 2021.

[29] Diego Doimo, Aldo Glielmo, Alessio Ansuini, and Alessandro Laio. Hierarchical nucleation in deep neural networks. *Advances in Neural Information Processing Systems*, 33:7526–7536, 2020.

[30] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[31] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[32] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018.

[33] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rJl-b3RcF7`.

[34] Jonathan Frankle, Gintare Karolina Dziugaite, and M Daniel. Stabilizing the lottery ticket hypothesis. *CoRR, abs*, 2019.

[35] Jonathan Frankle, David J Schwab, and Ari S Morcos. The early phase of neural network training. *arXiv preprint arXiv:2002.10365*, 2020.

[36] Kunihiko Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position-Neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979.

[37] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6):476–493, 2021.

[38] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88, 2019.

[39] Richard C Gerum, André Erpenbeck, Patrick Krauss, and Achim Schilling. Sparsity through evolutionary pruning prevents neuronal networks from overfitting. *Neural Networks*, 128:305–312, 2020.

[40] Gabriel Goh, Nick Cammarata, Chelsea Voss, Shan Carter, Michael Petrov, Ludwig Schubert, Alec Radford, and Chris Olah. Multimodal neurons in artificial neural networks. *Distill*, 2021. doi: 10.23915/distill.00030. https://distill.pub/2021/multimodal-neurons.

[41] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[42] Feng Guo, Yu Qian, and Yuefeng Shi. Real-time railroad track components inspection based on the improved yolov4 framework. *Automation in construction*, 125:103596, 2021.

[43] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[44] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[45] Haldan K Hartline. The receptive fields of optic nerve fibers. *American Journal of Physiology-Legacy Content*, 130(4):690–699, 1940.

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

[47] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 558–567, 2019.

[48] William Held and Diyi Yang. Shapley head pruning: Identifying and removing interference in multilingual transformers. *arXiv preprint arXiv:2210.05709*, 2022.

[49] Suzana Herculano-Houzel. The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *Proceedings of the National Academy of Sciences*, 109 (Supplement 1):10661–10668, 2012.

[50] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth

for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[51] Harold Hotelling. Relations between Two Sets of Variates. *Biometrika*, 28(3-4):321–377, 12 1936. ISSN 0006-3444. doi: 10.1093/biomet/28.3-4.321. URL `https://doi.org/10.1093/biomet/28.3-4.321`.

[52] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106, 1962.

[53] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[54] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[55] Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989.

[56] Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[57] Tong Jian, Zifeng Wang, Yanzhi Wang, Jennifer Dy, and Stratis Ioannidis. Pruning adversarially robust neural networks without adversarial examples. *arXiv preprint arXiv:2210.04311*, 2022.

[58] Mingjie Jiang, Xinqi Fan, and Hong Yan. Retinamask: A face mask detector. *arXiv preprint arXiv:2005.03950*, 2020.

[59] Zicong Jiang, Liquan Zhao, Shuaiyang Li, and Yanfei Jia. Real-time object detection method based on improved yolov4-tiny. *ArXiv*, abs/2011.04244, 2020.

[60] G Jignesh Chowdary, Narinder Singh Punn, Sanjay Kumar Sonbhadra, and Sonali Agarwal. Face mask detection using transfer learning of inceptionv3. In *International Conference on Big Data Analytics*, pages 81–90. Springer, 2020.

[61] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[62] Xiangjie Kong, Kailai Wang, Shupeng Wang, Xiaojie Wang, Xin Jiang, Yi Guo, Guojiang Shen, Xin Chen, and Qichao Ni. Real-time mask identification for covid-19: An edge computing-based deep learning framework. *IEEE Internet of Things Journal*, pages 1–1, 2021. doi: 10.1109/JIOT.2021.3051844.

[63] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International Conference on Machine Learning*, pages 3519–3529. PMLR, 2019.

[64] Alex Krizhevsky. The cifar-10 dataset, 2009. URL `https://www.cs.toronto.edu/~kriz/cifar.html`.

[65] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[66] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[67] Akhil Kumar, Arvind Kalia, Kinshuk Verma, Akashdeep Sharma, and Manisha Kaushal. Scaling up face masks detection with yolo on a novel dataset. *Optik*, 239:166744, 2021. ISSN 0030-4026. doi: https://doi.org/10.1016/j.ijleo.2021.166744.

[68] Mark Labbe. Energy consumption of ai poses environmental problems. *TechTarget*, 2021. URL `https://www.techtarget.com/searchenterpriseai/feature/Energy-consumption-of-AI-poses-environmental-problems`.

[69] Hung Le and Ali Borji. What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks? *arXiv preprint arXiv:1705.07049*, 2017.

[70] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[71] Byung-Kwan Lee, Junho Kim, and Yong Man Ro. Masking adversarial damage: Finding adversarial saliency for robust and sparse network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15126–15136, 2022.

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

[73] Ningyi Liao, Shufan Wang, Liyao Xiang, Nanyang Ye, Shuo Shao, and Pengzhi Chu. Achieving adversarial robustness via sparsity. *Machine Learning*, 111(2):685–711, 2022.

[74] Benedetta Liberatori, Ciro Antonio Mami, Giovanni Santacatterina, Marco Zullich, and Felice Andrea Pellegrino. Yolo-based face mask detection on low-end devices using pruning and quantization. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 900–905. IEEE, 2022.

[75] Tao Lin, Sebastian U. Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. In *International Conference on Learning Representations*, 2020.

[76] Grace W Lindsay. Convolutional neural networks as a model of the visual system: Past, present, and future. *Journal of cognitive neuroscience*, 33(10):2017–2031, 2021.

[77] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the vari-

ance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

[78] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Zahra Atashgahi, Lu Yin, Huanyu Kou, Li Shen, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. Sparse training via boosting pruning plasticity with neuroregeneration. *Advances in Neural Information Processing Systems*, 34:9908–9922, 2021.

[79] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rJlnB3C5Ym`.

[80] Lawrence K Low and Hwai-Jong Cheng. Axon pruning: an essential step underlying the developmental plasticity of neuronal connections. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 361(1473):1531–1544, 2006.

[81] Yunqian Ma and Yun Fu. *Manifold learning theory and applications*, volume 434. CRC press Boca Raton, 2012.

[82] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[83] Johannes Mehrer, Courtney J Spoerer, Nikolaus Kriegeskorte, and Tim C Kietzmann. Individual differences among deep neural network models. *Nature communications*, 11(1):1–12, 2020.

[84] Diganta Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, 4(2):10–48550, 2019.

[85] Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[86] Ari Morcos, Maithra Raghu, and Samy Bengio. Insights on representational similarity in neural networks with canonical correlation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman,

N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5732–5741. Curran Associates, Inc., 2018.

[87] Giorgia Nadizar, Eric Medvet, Felice Andrea Pellegrino, Marco Zullich, and Stefano Nichele. On the effects of pruning on evolved neural controllers for soft robots. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1744–1752, 2021.

[88] Giorgia Nadizar, Eric Medvet, Ola Huse Ramstad, Stefano Nichele, Felice Andrea Pellegrino, and Marco Zullich. Merging pruning and neuroevolution: towards robust and efficient controllers for modular soft robots. *The Knowledge Engineering Review*, 37, 2022.

[89] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[90] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *Advances in neural information processing systems*, 33:512–523, 2020.

[91] Thao Nguyen, Maithra Raghu, and Simon Kornblith. Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth. *arXiv preprint arXiv:2010.15327*, 2020.

[92] Beijing Academy of Artificial Intelligence. Wudaomm , 2021. URL `https://github.com/BAAI-WuDao/WuDaoMM`.

[93] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2017. doi: 10.23915/distill.00007. https://distill.pub/2017/feature-visualization.

[94] David C. Paige. cifar10-fast, 2018. URL `https://github.com/davidcpage/cifar10-fast`.

[95] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[96] Saini Pooja and Saini Preeti. Face mask detection using ai. In *Predictive and Preventive Measures for Covid-19 Pandemic*, pages 293–305. Springer, 2021.

[97] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[98] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. *Advances in neural information processing systems*, 30, 2017.

[99] Maithra Raghu, Thomas Unterthiner, Simon Kornblith, Chiyuan Zhang, and Alexey Dosovitskiy. Do vision transformers see like convolutional neural networks? *Advances in Neural Information Processing Systems*, 34:12116–12128, 2021.

[100] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017.

[101] Ingo Rechenberg. Evolutionsstrategien. In *Simulationsmethoden in der Medizin und Biologie*, pages 83–114. Springer, 1978.

[102] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[103] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[104] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[105] Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning. *arXiv preprint arXiv:2003.02389*, 2020.

[106] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[107] Biparnak Roy, Subhadip Nandy, Debojit Ghosh, Debarghya Dutta, Pritam Biswas, and Tamodip Das. Moxa: a deep learning based unmanned approach for real-time monitoring of people wearing medical masks. *Transactions of the Indian National Academy of Engineering*, 5(3):509–518, 2020.

[108] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323 (6088):533–536, 1986.

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

[110] Jun Sang, Zhongyuan Wu, Pei Guo, Haibo Hu, Hong Xiang, Qian Zhang, and Bin Cai. An improved yolov2 for vehicle detection. *Sensors*, 18(12):4272, 2018.

[111] Fadil Santosa and William W Symes. Linear inversion of band-limited reflection seismograms. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1307–1330, 1986.

[112] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[113] Nils T Siebel, Jonas Botel, and Gerald Sommer. Efficient neural network pruning during neuro-evolution. In *2009 International*

Joint Conference on Neural Networks, pages 2920–2927. IEEE, 2009.

[114] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[115] Congzheng Song and Vitaly Shmatikov. Overlearning reveals sensitive attributes. *arXiv preprint arXiv:1905.11742*, 2019.

[116] Olaf Sporns, Dante R Chialvo, Marcus Kaiser, and Claus C Hilgetag. Complex networks: small-world and scale-free architectures. *Trends in Cognitive Sciences*, 9(8):418–425, 2004.

[117] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[118] Suraj Srinivas, Andrey Kuzmin, Markus Nagel, Mart van Baalen, Andrii Skliar, and Tijmen Blankevoort. Cyclical pruning for sparse neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 2762–2771, June 2022.

[119] Computer Science Department Stanford University. The street view house numbers (svhn) dataset, 2011. URL `http://ufldl.stanford.edu/housenumbers/`.

[120] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[121] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Texts in Computer Science. Springer, 2 edition, 2022. ISBN 3030343715; 9783030343712.

[122] Shuai Tang, Wesley J Maddox, Charlie Dickens, Tom Diethe, and Andreas Damianou. Similarity of neural networks with gradients. *arXiv preprint arXiv:2003.11498*, 2020.

[123] Hans Henrik Thodberg. Improving generalization of neural networks through pruning. *International Journal of Neural Systems*, 1(04):317–326, 1991.

[124] Rob Toews. Deep learning's carbon emissions problem. *Forbes*, 2020. URL `https://www.forbes.com/sites/robtoews/2020/06/17/deep-learnings-climate-change-problem/`.

[125] Anton Tsitsulin, Marina Munkhoeva, Davide Mottin, Panagiotis Karras, Alex Bronstein, Ivan Oseledets, and Emmanuel Müller. The shape of data: Intrinsic distance for data distributions. *arXiv preprint arXiv:1905.11141*, 2019.

[126] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[127] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Association for Computational Linguistics (ACL)*, pages 5797–5808, 05 2019.

[128] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[129] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 13029–13038, 2021.

[130] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022.

[131] Tiancai Wang, Rao Muhammad Anwer, Hisham Cholakkal, Fahad Shahbaz Khan, Yanwei Pang, and Ling Shao. Learning rich features at high-speed for single-shot object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1971–1980, 2019.

[132] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[133] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[134] Shijie Wu, Alexis Conneau, Haoran Li, Luke Zettlemoyer, and Veselin Stoyanov. Emerging cross-lingual structure in pretrained language models. *arXiv preprint arXiv:1911.01464*, 2019.

[135] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[136] Guangxuan Xu and Qingyuan Hu. Can model compression improve nlp fairness. *arXiv preprint arXiv:2201.08542*, 2022.

[137] Shihui Yin, Kyu-Hyoun Kim, Jinwook Oh, Naigang Wang, Mauricio Serrano, Jae-Sun Seo, and Jungwook Choi. The sooner the better: Investigating structure of early winning lottery tickets. 2019.

[138] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 202.

[139] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[140] Fang Yu, Kun Huang, Meng Wang, Yuan Cheng, Wei Chu, and Li Cui. Width & depth pruning for vision transformers. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 2022, 2022.

[141] Jimin Yu and Wei Zhang. Face mask wearing detection algorithm based on improved yolo-v4. *Sensors*, 21(9):3263, 2021.

[142] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[143] Matthew D Zeiler, M Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, 2013.

[144] Byoung-Tak Zhang and Heinz Mühlenbein. Genetic programming of minimal neural nets usingOccam's razor. In *Proceedings of the 5th international conference on genetic algorithms (ICGA'93*. Citeseer, 1993.

[145] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, pages 3597–3607, 2019.

[146] Marco Zullich, Eric Medvet, Felice Andrea Pellegrino, and Alessio Ansuini. Speeding-up pruning for artificial neural networks: introducing accelerated iterative magnitude pruning. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 3868–3875. IEEE, 2021.

[147] Marco Zullich, Vanja Macovaz, Giovanni Pinna, and Felice Andrea Pellegrino. An artificial intelligence system for automatic recognition of punches in fourteenth-century panel painting. *IEEE Access*, 2022. Under Review.

About the author

Marco Zullich was born in Trieste (Italy) in 1990. From 2009 to 2013 he studied Statistics at the University of Trieste, gaining his B.Sc. in 2012. He then worked for one year as a market researcher at SWG S.p.A, still in Trieste; then was for five years an insurance analyst for Capgemini and Allianz. In 2017, he resumed his academic career, completing a M.Sc. in Data Science and Scientific Computing at the University of Trieste in 2019. He subsequently started, and is now in the process of completing, a Ph.D. in Industrial and Information Engineering at the University of Trieste. His research interests include Machine Learning, specifically Deep Learning, a subject of which he was teaching assistant during his Ph.D. He has worked on pruning and model compression, but also on the application of Machine Learning techniques to the field of cultural heritage.

List of publications

Alessio Ansuini, Eric Medvet, Felice Andrea Pellegrino, and **Marco Zullich**. On the similarity between hidden layers of pruned and unpruned convolutional neural networks. In ICPRAM, pages 52–59, 2020.

Alessio Ansuini, Eric Medvet, Felice Andrea Pellegrino, and **Marco Zullich**. Investigating similarity metrics for convolutional neural networks in the case of unstructured pruning. In International Conference on Pattern Recognition Applications and Methods, pages 87–111. Springer, 2020.

Marco Zullich, Eric Medvet, Felice Andrea Pellegrino, and Alessio Ansuini. Speeding-up pruning for artificial neural networks: introducing accelerated iterative magnitude pruning. In 2020 25th International Conference on Pattern Recognition (ICPR), pages 3868–3875. IEEE, 2021.

Giorgia Nadizar, Eric Medvet, Felice Andrea Pellegrino, **Marco Zullich**, and Stefano Nichele. On the effects of pruning on evolved neural controllers for soft robots. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pages 1744–1752, 2021.

Giorgia Nadizar, Eric Medvet, Ola Huse Ramstad, Stefano Nichele, Felice Andrea Pellegrino, and **Marco Zullich**. Merging pruning and neuroevolution: towards robust and efficient controllers for modular soft robots. The Knowledge Engineering Review, 37, 2022.

Benedetta Liberatori, Ciro Antonio Mami, Giovanni Santacatterina, **Marco Zullich**, and Felice Andrea Pellegrino. Yolo-based face mask detection on low-end devices using pruning and quantization. In 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO), pages 900–905. IEEE, 2022.

Marco Zullich, Vanja Macovaz, Giovanni Pinna, and Felice Andrea Pellegrino. An artificial intelligence system for automatic recognition of punches in fourteenth-century panel painting. IEEE Access, 2022. *Under Review*.