UNIVERSITÀ DEGLI STUDI FIRENZE

PhD Program in Smart Computing
Dipartimento di Ingegneria dell'Informazione (DINFO)

Neural Architecture Search by Growing Internal Computational Units

Vincenzo Laveglia

Dissertation presented in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Smart Computing

PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Neural Architecture Search by Growing Internal Computational Units

Vincenzo Laveglia

Advisor:

Prof. Edmondo Trentin

Head of the PhD Program:

Prof. Paolo Frasconi

Evaluation Committee:
Prof. Friedhelm Schwenker, *Ulm University, Germany*
Prof. Hazem Abbas, *Ain Shams University, Cairo, Egypt*

XXXI ciclo — October 2019

Acknowledgments

First I want to thank my advisor, Edmondo Trentin, who conveyed me the passion for this topic and stimulated with his discourses. He has been a friend and an inspiration. I thank him for guiding me along this path.

I am grateful to all the Professors and Researchers of the Artificial Intelligence Group of the University of Siena for all the very formative discussions and support. A special thanks to Franco Scarselli and Monica Bianchini.

A huge acknowledge goes to all my friends and colleagues that shared with me these years at the Artificial Intelligence Lab. In particular to Alessandro Rossi, Francesco Giannini, Dario Zanca and Andrea Zugarini, whom I shared ideas and intuitions, providing me with important feedbacks and suggestions. Their support has been invaluable.

I wish also to express my gratitude to Roldano Cattoni and Stefan Scherer for having supervised my work every year, and for their precious indications.

I am deeply grateful to my family for their loving support.

Thanks to Cecilia, for her patience throughout these years.

Finally, thanks to University of Florence for granting me the PhD fellowship that has made possible the pursuing of this goal.

Abstract

Choosing the right neural network architecture for a given learning task is still an open problem. It is well known among practitioners that it often requires several trials, with a consequent wasting of lot of time. It is an active research area, and several solutions have been proposed in last years. Anyway, the nowadays common practise is to set a certain architecture and than update its configuration (network weights) in order to achieve good performances. Furthermore, over the search for the right number of neurons and layers, also other components, as the activation functions, need to be considered in the choice of the right neural architecture. The aim of this research is to go over the classical concept of learning model, where the architecture is static and the model performances are evaluated only at the whole model level. The idea is to design a learning model that can autonomously define an its own internal structure, and with the capacity to discover the particular components of its architecture that need to be empowered, avoiding in this way to impact the whole model configuration. The steps taken in achieving this goal have gone through the development of the following three milestones: *Target Propagation*, *Depth-Growing Neural Networks* and *Downward-Growing Neural Architectures*; which also reflect the organization of the Thesis.
Here we define the *Depth Growing Neural Network* framework (DGNN), as having the following main features: first, a dynamic architecture, it evolves during the learning process in order to autonomously find the internal structure that best suits the needed computational power; second, the evolution process is driven by an evaluation of the single components of its architecture, the so called *metaneurons*; their performances are rated with regards to their expected outcomes, and the worst performing metaneurons are chosen to be upgraded. Adhoc algorithms have been developed to estimate metaneuron outcomes, these are identified as *Target Propagation* techniques.
The mentioned evolution process basically consists in the transformation of a single internal neurons, or a set of them, in a more complex structure, as a set of interconnected neurons; obtaining in this way more powerful computational units. We define these more complex structures as *subnets*. The internal neurons of the *subnets* can evolve in turn, realizing in this way a recursive process that can lead to the building of deep architectures. The definition of the right setting for these new structures is acted using a classical gradient descent approach, by back propagating the errors with respect to the estimated outcomes of the subnets.
As mentioned above, few techniques have been developed for estimating the metaneuron (and subnet) outcomes. These consist in propagating the neural network target outputs to the internal layers of the network, defining in this way layer-specific targets, allowing to formalize a layer-specific and potentially neuron-specific loss function.

Although DGNN can represent a step through the development of autonomously defined architectures, empirical evidence highlighted some of their limitations, first of all the difficulty for subnets to learn their given learning tasks. This was basically a consequence of how the DGNN architecture was conceived.
The solutions designed to overcome limitations highlighted in the DGNN, resulted in the development of the *Downward-Growing Neural Architecture* (DGNA) framework. The latter indirectly shares the same goal and philosophy of its ancestor, but implements a completely different growing strategy. Here the evolution of the architecture is seen as a consequence of another goal, that is to improve the performances of the model by defining more and more complex decision regions. This is realized by replacing computational units, that define a certain decision region, with more powerful computational components. An in depth analysis of this topic is carried on in the Thesis, identifying as a resulting solution the replacement of hidden neurons and their input connections with brand new subnets, one for each hidden neuron. This approach entails a substantial modification of the first layer of the network, contrariwise to what happens with the previous model. Experimental results validate this technique.

Contents

Contents 1

List of Figures 3

List of Tables 6

1 Introduction 9

2 Target Propagation 13
- 2.1 Definitions 15
- 2.2 Existing Methods 16
- 2.3 Error Driven Target Propagation 19
- 2.4 Residual Driven Target Propagation 24
- 2.5 Gradient Based - Residual Driven Target Prop 28
- 2.6 Experiments with the Refinement Algorithm 30
- 2.7 Remarks 34

3 Depth Growing Neural Networks 35
- 3.1 Related Works (adaptive activation functions) 36
- 3.2 The DGNN Model 37
- 3.3 Training Algorithm 40
- 3.4 Parallelizing the Algorithm 45
- 3.5 Experimental Results 47
- 3.6 Remarks 72

4 Downward-Growing Neural Architectures 73
- 4.1 Related Works 74
- 4.2 Growing Architectures as a Search Strategy 77
- 4.3 The Learning Algorithm 81
- 4.4 Experimental results 86
- 4.5 Remarks 98

5 Conclusions 99

A Publications **101**

Bibliography **103**

List of Figures

2.1 An illustration of how targets propagate downward through the network. 24
2.2 Learning and generalization curves for *dnet*. 31
2.3 Learning and generalization curves of the procedure *layer_backprop*(.) applied to the three hidden layers of *dnet*. 32
2.4 Learning and generalization curves of the procedure *layer_backprop*(.) applied to the output layer of *dnet*. 32

3.1 Representation of a DGNN having depth $k = 0$. The meta-neurons are represented with a custom circular shape. On the left side are indicated the mapping functions associated to the single layers. 40
3.2 Representation of a DGNN having depth $k = 1$ (after a growing step). The meta-neurons at depth $k = 0$ have been replaced by subnets $S_1, ..., S_4$ having depth $k = 1$. Here all meta-neurons have been replaced, and all subnets have the same architecture. 41
3.3 Two subnets $S_1^{[k]}, S_2^{[k]}$ are represented as being part of two consecutive standard layer, where the dotted lines represent the connections having fixed weights $w = 0$ 44
3.4 Vertebral, depth-0 49
3.5 Vertebral, depth-1 50
3.6 Vertebral, depth-1 subnets 50
3.7 Vertebral, depth-2 50
3.8 Vertebral, depth-2 subnets 50
3.9 Vertebral, depth-3 51
3.10 Vertebral, depth-3 subnet 51
3.11 Vertebral, depth-0 acc. 51
3.12 Vertebral, depth-1 acc. 51
3.13 Vertebral, depth-2 acc. 52
3.14 Vertebral, depth-3 acc. 52
3.15 Vertebral, depth-0 53
3.16 Vertebral, depth-1 53
3.17 Vertebral, depth-1 subnets 53

3.18 Vertebral, depth-2 54
3.19 Vertebral, depth-2 subnets 54
3.20 Vertebral, depth-3 54
3.21 Vertebral, depth-3 subnets 54
3.22 Vertebral, depth-0 acc. 54
3.23 Vertebral, depth-1 acc. 54
3.24 Vertebral, depth-2 acc. 55
3.25 Vertebral, depth-3 acc. 55
3.26 Vertebral 10-th fold,
depth-0 acc. 56
3.27 Vertebral 10-th fold,
depth-1 acc. 56
3.28 Vertebral 10-th fold,
depth-2 acc. 56
3.29 Vertebral 10-th fold,
depth-3 acc. 56
3.30 MNIST depth-0 loss 59
3.31 MNIST depth-1 loss 59
3.32 MNIST depth-1 subnet's loss 59
3.33 MNIST depth-2 loss 59
3.34 MNIST depth-2 subnet's loss 59
3.35 MNIST depth-0 acc. 60
3.36 MNIST depth-1 acc. 60
3.37 MNIST depth-2 acc. 60
3.38 Deer-Truck depth-0 loss 62
3.39 Deer-Truck depth-1 loss
. 63
3.40 Deer-Truck depth-1 subnet's loss 63
3.41 Deer-Truck depth-2 loss
. 63
3.42 Deer-Truck depth-2 subnet's loss 63
3.43 Deer-Truck depth-0 acc. 64
3.44 Deer-Truck depth-1 acc. 64
3.45 Deer-Truck depth-2 acc. 64
3.46 Deer-Horse depth-0 loss 65
3.47 Deer-Horse depth-1 loss
. 66
3.48 Deer-Horse depth-1 subnet's loss 66
3.49 Deer-Horse depth-2 loss
. 66

3.50 Deer-Horse depth-2 subnet's loss 66
3.51 Deer-Horse depth-0 acc. 67
3.52 Deer-Horse depth-1 acc. 67
3.53 Deer-Horse depth-2 acc. 67
3.54 Car-Dog depth-0 loss 68
3.55 Car-Dog depth-1 loss
. . . . 69
3.56 Car-Dog depth-1 subnet's loss 69
3.57 Car-Dog depth-2 loss
. . . . 69
3.58 Car-Dog depth-2 subnet's loss 69
3.59 Car-Dog depth-0 acc. 70
3.60 Car-Dog depth-1 acc. 70
3.61 Car-Dog depth-2 acc. 70

4.1 This image visually describes what we expect from a growing model. We have a set of points $\in \mathbb{R}^2$ belonging to two classes (cross and circles). We assume a single hidden layer MLP as classifier. **Left**: separation surfaces defined by two hidden neurons. **Right**: separation surfaces expected to be generated by a growing model, where o_1' is the "evolved version" of o_1. 79
4.2 **Left**: standard 1 hidden layer feedforward network (FFN), or base-network. **Right**: the grown network, after replacing the leftmost neuron and its input connection with a subnet. 81

List of Tables

2.1 Accuracy on MNIST 10-fold classification task (avg. ± std. dev. on a 10-fold crossvalidation) 30
2.2 Comparison between the proposed algorithm and the established approaches, in terms of error rate and number of adaptive parameters. . . . 30

3.1 Vertebral 10-fold cross-validation acc. ± std 57
3.2 MNIST 5-folds crossvalidation accuracy. 58
3.3 MNIST 5-folds crossvalidation MSE loss. 58
3.4 Deer-Truck 5-folds crossvalidation acc. ± std. dev. 62
3.5 Deer-Truck 5-folds crossvalidation MSE loss ± std. dev. 62
3.6 Deer-Horse 5-folds crossvalidation acc. ± std. dev. 65
3.7 Deer-Horse 5-folds crossvalidation MSE loss ± std. dev. 65
3.8 Car-Dog 5-folds crossvalidation acc. ± std. dev. 68
3.9 Car-Dog 5-folds crossvalidation MSE loss ± std. dev. 68

4.1 Characteristics of the datasets used to validate our model. 86
4.2 DGNA considered hyperparameters 87
4.3 Accuracy values for the different growing sub-steps and relative average improvement. 88
4.4 MSE loss values for different growing sub-steps and relative average improvement. 88
4.5 Accuracy values for different growing sub-steps and relative average improvement. 89
4.6 MSE loss values for different growing sub-steps and relative average improvement. 89
4.7 Accuracy values for different steps. and relative improvement. 90
4.8 MSE loss values for different steps. and relative improvement. 90
4.9 Accuracy values for different steps. 91
4.10 MSE loss values for different steps. 91
4.11 Accuracy values for different sub-steps. 92
4.12 MSE loss values for different sub-steps. 92
4.13 Accuracy values for different steps. 93

4.14 MSE loss values for different steps. 93
4.15 Accuracy values for different growing steps. 94
4.16 MSE loss values for different growing steps. 94
4.17 Considered hyperparameters 96
4.18 Accuracy comparison for different models trained with the UCI datasets 96

Chapter 1

Introduction

Despite the recent success of neural networks in being extremely performing in most of the application fields, they were left behind the scene for many years. Artificial neural networks (ANNs), were originally defined in [42], in the middle of last century. After the *first AI winter*, started in the end of 60s, when Minsky and Papert identified few computational issues [43], stating that these could not be properly trained, ANNs came back to the light in the middle of 80s, when the *backpropagation* (BP) training algorithm was defined [56, 49], making them usable. As it often happens, history is cyclic, and in the middle of 90s, the scientific community begun again to pay less attention to ANNs, because of their limitations. In the end of 00s, few of the drawbacks that let neural networks fall in this second AI winter (end of 90s - beginning of 00s), as the difficulty of training deep architecture and the high computing power required, were partially solved. The development of graphic cards, the availability of always bigger datasets, and new training techniques like layer-wise training [4], gave birth to the *renaissance* of neural networks, in their new form of deep architecture or deep neural networks (DNN) [3]. Other works, like that of dropout [52] and the introduction of ReLU [24] affirmed this new trend.

It has been seen that ANNs work outstandingly even if dealing with input patterns of huge size and at predicting thousands of classes. Also emerged that deep architectures extract very interesting data representations in a hierarchical fashion for each of its internal layers. It deserves to be said that DNNs set foot in all the main areas, like computer vision, speech recognition, natural language processing/understanding, reaching state of the art results in most of them. Lastly, it has been seen that if set in a specific configuration and trained using a particular procedure, said *generative adversarial* [25], DNNs can work as generative models, obtaining impressive results. From these considerations, it is clear that if properly used, DNNs can be very useful to tackle most problems. However it must be said that optimize such models is not trivial at all.

It is well known among practitioners that training a DNN can potentially results in very burdensome experiences. Apart of the backpropagation algorithm to barely execute, lot of aspects must to be taken into account. Lot of experience is required. Given a task $\mathcal{T}$ represented with the dataset $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^{N}\}$, where $\mathbf{x}_j \in \mathbb{R}^n$ and $\hat{\mathbf{y}}_j \in \mathbb{R}^m$, the first step in creating a DNN able to tackle $\mathcal{T}$ is the setting of the input and output size, that correspond to the sizes of the generic input pattern and output target, in this case n and m. The second step is the choice of activation functions for the output layer. This is not an arbitrary choice. Output functions codomain has to be such that it will contains the target values. The third step is the choice of internal structure of the network, on which the model's performances mainly depend. The main aspects to consider are: the number of layers, the size of each layer (often all internal layers have the same size), that greatly affect the computational capabilities of the network. Lastly, the choice of activation functions for hidden layers. If dealing with many layers, quasi-linear activation functions (ReLu [24], Elu [12], etc.) are advisable, in order to overcome common issues related to the nature of the BP training algorithm [23]. Once the architectural choices have been made, a learning strategy must be implemented. This entails the mini-batch size, a drop-out value [52], an eventual batch-normalization [29], and other common used features. As it is, the choice of the best architecture and learning strategy is not a deterministic phenomena, but most of the time it is the outcome of a long search strategy. It is demonstrated by the fact that hyperparameters optimization is an active research area. Several complex hyper-parameter selection algorithms have been developed by the scientific community, and some of them in turn relies on gradient descent strategy [21]. To be able to manage all these describes tools, and understanding the related consequence, experience is required. The latter practise can be considered a subfield of what is known as AutoML [28], that aims at defining more and more automated procedures to engineer machine learning systems. AutoML also covers the topic of neural architecture search (NAS) [18], which this Thesis mainly deal with. Although we place this work in the NAS field, some slight differences exists. The here proposed algorithms do not perform a "search" operation in the strict sense, (as most works mentioned in [18]), rather they aim to bring out the neural architecture as a consequence of a network growth process.

Chapter 2 of the thesis deals with concept of *Target Propagation*. Here some affine and related techniques are discussed, then a detailed description of the new proposed algorithms is held. Furthermore, one of these algorithms is used as a tool in the definition of a technique aimed at refining pre-trained neural networks. This is extensively explained, and few experimental results are given.
In chapter 3 the *Depth Growing Neural Network* framework (DGNN) is introduced. Here it is formalized, its training algorithm is described and an optimization strategy for its implementation is introduced. In the last section of the chapter experi-

mental results are reported. The model has been tested on the *vertebral column dataset* of UCI machine learning repository [16], the MNIST dataset, and 3 sub-problems of CIFAR-10 dataset [35] . Experiments have been designed to show the ability of the model to act as expected, and to confirm its behaviour. First two experiments on UCI Vertebral report the learning curves and the accuracy for two specific runs, showing different performance behaviours at varying the subnet size hyperparameter; while last experiment is a 10-fold cross validation process, aimed at reporting the grade of stability of performances. Other experiments on bigger dataset as MNIST and CIFAR-10 are intended to confirm what emerged with the UCI dataset.
Even if seems promising, to the state of the art, expectations on the DGNN model have only partially been verified.
In chapter 4 an extensive discussion of the idea behind the *Downward-Growing Neural Architecture* (DGNA) is carried on. The learning algorithm is described, and an experimental section on 7 UCI dataset is performed. It worth to say that results assess the effectiveness of the proposed algorithm, and a comparison section with other results state that is it is in line with other well established deep network models and often even considerably better.

Contributions:
The work done in the development of this Thesis is completely original. Each of the chapters introduces new definitions and algorithms that are the outcome of the research activities pursued during the PhD.
The contribution of this Thesis to the topic of "neural architecture search" can be summarized in the following points:

- The definition of several *Target Propagation* algorithms.
- The definition of the *Depth-Growing Neural Network* model (DGNN).
- The definition of the *Downward-Growing Neural Architecture* framework (DGNA), and the assessment of its effectiveness as an alternative neural architecture search tool.

Chapter 2

Target Propagation

Artificial neural networks, are optimized using a three-decades old algorithm, commonly known as *backpropagation* (BP) [49], that in this work we will also remind as standard backpropagation, to not confuse with other similar algorithms that will be defined in next sections.
The idea behind BP is that each weight of the network is partially accountable for the output error of the model, regarding the particular input-target pair $(\mathbf{x}, \hat{\mathbf{y}})$; it works as in the following. In order to reduce the output error, weights of network are updated w.r.t. to how that particular parameter influenced or could potentially influence the decision of the network. For the generic weight w_i, this corresponds to the calculation of the partial derivative of the the loss function $\mathcal{L}$ w.r.t the generic weight w_i, $\frac{\partial}{\partial w_i}\mathcal{L}$. The weights are then updated as in

$$w_i' = w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i} \tag{2.1}$$

where η is the *learning rate* i.e. the size of the step that we take in the direction of the gradient. Eq. 2.1 corresponds to find a new network configuration associated to a smaller loss function value. It is basically done moving the weights through the hyperspace in the direction indicated by the negative of the gradient of $\mathcal{L}$ with respect to the weights of the model. Bearing in mind the architecture of a multi-layer neural network, the calculation of the gradient with respect to the weights of internal (or hidden) layers, is done using the chain rule of the derivative. While standard backpropagation works outstandingly on networks having a limited number of hidden layers, several weaknesses of the algorithm emerge when dealing with significantly deep architectures [23]. In particular, due to the non-linearity of the activation functions associated to the units in the hidden layers, the backpropagated gradients tend to vanish in the lower layers of the network, hence hindering the corresponding learning process. Besides its numerical problems, BP is also known to lack any plausible biological interpretation [5]. To overcome these difficulties, researchers proposed improved learning strategies, such as pre-training of the lower

layers via auto-encoders [4], the use of rectifier activation functions [24], and the dropout technique [52] to avoid neurons co-adaptation.

Amongst these and other potential solutions to the aforementioned difficulties, target propagation has been arousing interest in the last few years [39, 10], albeit it still remains an under-investigated research area. Originally proposed in [8, 9] within the broader framework of learning the form of the activation functions, the idea underlying target propagation goes as follows.
While in BP the signals to be back propagated are related to the partial derivatives of the global loss function w.r.t. the layer-specific parameters of the network, in target propagation the real target outputs (naturally defined at the output layer in regular supervised learning) are propagated downward through the network, from the topmost to the bottommost layers. In so doing, each layer gets explicit target output vectors that, in turn, define layer-specific loss functions that can be minimized locally (on a layer by layer basis) without any need to involve explicitly the partial derivatives of the overall loss function defined at the whole network level. Therefore, the learning process gets rid altogether of the troublesome numerical problems determined by repeatedly backpropagating partial derivatives from top to bottom.

The goal of this chapter is to discuss a new potential neural network training technique/framework, said *target propagation*. The main idea here is to determine target values for each neuron (and layer) of the network. Analogously to how standard backpropagation works, this targets are first calculated for the topmost layer and using a propagation technique, new targets are determined for the internal layers, in a top-bottom fashion.

2.1 Definitions

Given a neural network *dnet* having ℓ layers, given the generic output target $\hat{\mathbf{y}}_\ell$ of the network, which is associated to the ℓ-th layer (the output layer), *target propagation* consists in estimating the target value $\hat{\mathbf{y}}_{\ell-1}$ for layer $\ell - 1$. In order to accomplish the task, a specific function $\phi(.)$ has to be realized, such that

$$\hat{\mathbf{y}}_{\ell-1} = \phi(\hat{\mathbf{y}}_\ell) \tag{2.2}$$

When *dnet* is fed with an input vector $\mathbf{x}$, the i-th layer of *dnet* (for $i = 1, ..., \ell$, while $i = 0$ represents the input layer which is not counted) is characterized by a state $\mathbf{h}_i \in \mathbb{R}^{d_i}$, where d_i is the number of units in layer i, $\mathbf{h}_i = \sigma(W_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ and $\mathbf{h}_0 = \mathbf{x}$ as usual. The quantity W_i represents the weight matrix associated to layer i, $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$, $\mathbf{b}_i \in \mathbb{R}^{d_i}$ denotes the corresponding bias vector and $\sigma_i(.)$ represents the vector of the element-wise outcomes of the activation functions for the specific layer i, but most of the times the layer index is omitted when it is not needed. (The logistic sigmoid activation function is used in the present research).
Consider a supervised dataset $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j) | j = 1, ..., N\}$. Given a generic input pattern $\mathbf{x} \in \mathbb{R}^n$ and the corresponding target output $\hat{\mathbf{y}} \in \mathbb{R}^m$ both drawn from $\mathcal{D}$, the state $\mathbf{h}_0 \in \mathbb{R}^n$ of the input layer of *dnet* is defined as $\mathbf{h}_0 = \mathbf{x}$, while the target state $\hat{\mathbf{h}}_\ell \in \mathbb{R}^m$ of the output layer is $\hat{\mathbf{h}}_\ell = \hat{\mathbf{y}}$. Relying on this notation, it is seen that the function $f_i(.)$ realized by the generic i-th layer in *dnet* can be written as

$$f_i(\mathbf{h}_{i-1}) = \sigma(W_i \mathbf{h}_{i-1} + \mathbf{b}_i) \tag{2.3}$$

Therefore, the mapping $F_i : \mathbb{R}^n \to \mathbb{R}^{d_i}$ realized by the i bottommost layers of *dnet* over current input $\mathbf{x}$ can be expressed as the composition of the i layer-specific functions as follows:

$$F_i(\mathbf{x}) = f_i(f_{i-1}...(f_1(\mathbf{x}))) \tag{2.4}$$

Eventually, the function realized by *dnet* (that is ℓ-layer network) is $F_\ell(\mathbf{x})$. Bearing in mind the definition of $\mathcal{D}$, the goal of training *dnet* is having $F_\ell(\mathbf{x}_j) = \hat{\mathbf{y}}_j$ for $j = 1, ..., N$. This is achieved by minimizing a point-wise loss function measured at the output layer. In the work done, this loss is the classic squared error represented as $\mathcal{L}(\mathbf{x}_j; \theta) = ||F_\ell(\mathbf{x}_j) - \hat{\mathbf{y}}_j||_2^2$, where θ represents the overall set of the parameters of *dnet* and $||\cdot||_2$ is the euclideian norm. In the traditional supervised learning framework the targets are defined only at the output layer. Nevertheless, while no explicit "loss" functions are associated to the hidden layers, the backpropagation (BP) algorithm allows the update of the hidden layers weights by back-propagating the gradients of the top level loss $\mathcal{L}(.)$. To the contrary, *target propagation* consist in propagating the topmost layer targets $\hat{\mathbf{y}}$ to the lower layers, in order to obtain explicit targets for the hidden units of the network, as well. Eventually, standard gradient descent with no BP is applied in order to learn the layer-specific parameters as a function of the corresponding targets.

2.2 Existing Methods

As already said, target propagation (TP) is still an under investigated research area, although several works exists in the literature. Here we roughly show some of the main works that most relate to the topic of this research. A first attempt to go in the direction of target propagation was made in the middle of 80s [37], although using threshold based (binary valued) architectures, where each neuron could only take values $\{-1, 1\}$. In such a way, estimating the target was basically equivalent to estimating the sign. Here we mention the TP methods based on the pseudoinverse and gradient descent algorithms introduced in [10], an affine approach said synthetic gradient described in [31] and the autoencoder based TP technique found in [39].

Pseudoinverse Based Target Prop

Given a network having ℓ layers, using the notation introduced in the previous section, the generic network output can be written as $\mathbf{y} = \sigma(W_\ell \mathbf{h}_{\ell-1} + \mathbf{b}_\ell)$, where $\sigma(.)$ is the non-linear activation function of the output layer. Consequently, the generic target output value can be written as $\hat{\mathbf{y}} = \sigma(W_\ell \hat{\mathbf{h}}_{\ell-1} + \mathbf{b}_\ell)$. Looking at the latter, the first idea the can be conceived in order to find the targets for layer $\ell - 1$, is that of mathematically explicit $\hat{\mathbf{h}}_{\ell-1}$ from the equation. So we can think to propagate the target to layer $\ell - 1$ as in the following:

$$\hat{\mathbf{h}}_{\ell-1} = W_\ell^{-1}(\sigma^{-1}(\hat{\mathbf{y}}) - \mathbf{b}_\ell). \tag{2.5}$$

Unfortunately, this equation can be solved only is the matrix W_ℓ is invertible. Among the conditions required for a matrix to be invertible, it is necessary that W_ℓ be a square matrix; this basically entails that layers $\ell - 1$ and ℓ have to be of the same size, which is not a very conventional neural network architecture. To overcome this issue, an inversion technique that deals with non-square matrix can be used, as the Moore-Penrose pseudo-inverse [47], whose mathematical description goes over the scope of this work. This approach has been used in [10], anyway this is just an approximation, and the bigger is the difference between the size of layers ℓ and $\ell - 1$, the bigger is the approximation error, often leading to numerical problems and instability. Basically this approach can be used only if the network architecture respects the conditions described above, or if the difference between layers size is negligible. Furthermore, the size of matrix to invert should also be considered, due to the expensive computational cost of the inversion operation.

Gradient Based Target Prop

In [10], the author describes a gradient based technique aimed at the estimation of target values for layer $\ell - 1$. Having $\mathbf{y} = \sigma(W_\ell \mathbf{h}_{\ell-1} + \mathbf{b}_\ell)$ and $\hat{\mathbf{y}} = \sigma(W_\ell \hat{\mathbf{h}}_{\ell-1} + \mathbf{b}_\ell)$

as before, for the estimation of $\hat{\mathbf{h}}_{\ell-1}$ the author aims at minimizing a loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$, where $\mathbf{y}$ is the output realized by feeding the ℓ-th layer with the newly generated $\hat{\mathbf{h}}_{\ell-1}$, who is here estimated using the following update rule

$$\hat{\mathbf{h}}_{\ell-1} = \hat{\mathbf{h}}_{\ell-1} - \eta \frac{\partial \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})}{\partial \hat{\mathbf{h}}_{\ell-1}} \tag{2.6}$$

where the updating continues until a stopping policy is met. As it has been defined, its computational cost scales linearly with the number of output targets to invert. This makes it not usable when dealing with large dataset. To overcome this issue, target estimation can be performed in batch or mini-batch mode.

Synthetic Gradient

A very interesting work, loosely related to the topic of this section, but with few things in common, is the one proposed in [31]. It was conceived with the goal of being able to train the multiple layers of a generic network all in parallel, without waiting for the gradients of the global loss function to be propagated back through the upper layers. To do that, for each layer i, with $i = 1, ..., \ell - 1$, a specific neural network net_i is defined, aimed at learning to predict the value of the gradient that will be back propagated to layer i by the upper layers, given the outcome of the i-th layer. Basically this approach is analogous to TP, but it estimates the layer-specific gradients instead of the targets. What make it interesting is the auxiliary layer-specific network, that has also been used in a recent TP algorithm, described in the following paragraphs.

Difference Target Propagation

A recent milestone regarding the target propagation topic, achieved using common continuous valued neural network architecture, is the one described in [39]. Here for each layer of the network, targets are estimated using an autoencoder like approach. To this end, [39] proposed an approach called difference target propagation (DTP) that relies on autoencoders. DTP is aimed at realizing a straight mapping $\hat{\mathbf{y}}_{\ell-1} = \phi(\hat{\mathbf{y}}_\ell)$ from the targets $\hat{\mathbf{y}}_\ell$ at layer ℓ to the expected targets $\hat{\mathbf{y}}_{\ell-1}$ at layer $\ell - 1$. The main idea is to realize a function $g_i(.)$ associated to every layer-specific function $f_i(.)$ as $\mathbf{h}_{i-1} = g_i(\mathbf{h}_i)$ and such that $f_i(g_i(\mathbf{h}_i)) \simeq \mathbf{h}_i$. This function $g_i(.)$ is seen as the decoding part of a more generic auto-encoder, that can be formalized as the composition of functions $g_i(f_i(.))$. This decoding part is trained in order to minimize a layer-specific loss $\mathcal{L}_i = \|g_i(f_i(\mathbf{h}_{i-1})) - \mathbf{h}_{i-1}\|_2^2$, so that the learning process will make $g_i(.) \simeq f_i^{-1}(.)$. As it is, it estimates the inverse mapping. In order to be able to estimate the inverse mapping also for target values, such that $\hat{\mathbf{h}}_{i-1} = g_i(\hat{\mathbf{h}}_i)$, thinking to target values as data points laying in the neighbourhood of the training set points, the loss is update by introducing a random noise

as $\mathcal{L}_i = \|g_i(f_i(\mathbf{h}_{i-1} + \epsilon)) - (\mathbf{h}_{i-1} + \epsilon)\|_2^2$, where $\epsilon \sim N(0, \sigma)$, who is aimed at learning to estimate the inverse mapping for neighbours $\mathbf{h}_{i-1} + \epsilon$, who hopefully are very close to target points $\hat{\mathbf{h}}_{i-1}$. As shown by [39], the technique is effective (it improves over regular gradient-descent in the experiments carried out on the MNIST dataset), although the accuracy yielded by DTP does not compare favorably with the state-of-the-art methods (mostly based on convolutional networks). Moreover, DTP offers the advantages of being readily applied to stochastic and discrete neural nets.

2.3 Error Driven Target Propagation

Differently from DTP, the core of the present approach is that the backward mapping from layer ℓ to $\ell - 1$ shall be learnt by a regular feed-forward neural network as an explicit function $\varphi(.)$ of the actual error $\mathbf{e}_\ell = \hat{\mathbf{y}}_\ell - \mathbf{y}_\ell$ observed at layer ℓ (namely, the signed difference between the target and actual outputs at ℓ), that is $\hat{\mathbf{y}}_{\ell-1} = \varphi(\hat{\mathbf{y}}_\ell, \mathbf{e}_\ell)$. In so doing, after training has been completed, the image of $\varphi(\hat{\mathbf{y}}_\ell, \mathbf{0})$ is an estimated optimal value of $\hat{\mathbf{y}}_{\ell-1}$ that is expected to result in a null error $\mathbf{e}_\ell = \mathbf{0}$ when propagated forward ($\hat{\mathbf{y}}_\ell = f_\ell(\varphi(\hat{\mathbf{y}}_\ell, \mathbf{0}))$ i.e. from $\ell - 1$ to ℓ) through the original network. It is seen that learning $\varphi(.)$ requires that at least a significant fraction of the training samples results in small errors (such that $\mathbf{e}_\ell \simeq \mathbf{0}$). This is the reason why the proposed technique can hardly be expected to be a suitable replacement for the established learning algorithms altogether, but it rather results in an effective refinement method for improving significantly the models realized by pre-trained deep neural networks. The proposed approach is different from that introduced in [4, 3], as well, since the latter relies on gradient-descent (or, the pseudo-inverse method) and, above all, it does not involve $\mathbf{e}_\ell$.
In this research, at the core of the target propagation algorithm there is another, subsidiary network called the *inversion net*. Its nature and its application to target propagation are handed out in the following section.

The Inversion Net

Let us assume that the target value $\hat{\mathbf{h}}_i$ is known for a certain layer i (eg. for the output layer, in the first place). The inversion net is then expected to estimate the targets $\hat{\mathbf{h}}_{i-1}$ for the preceding layer, that is layer $i - 1$. In this research the inversion net is a standard feed-forward neural network having a a much smaller number of parameters than *dnet* has, e.g. having a single hidden layer. In principle, as in [39], the inversion net could be trained such that it learns to realize a function $g_i() : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_{i-1}}$ defined as

$$g_i(\hat{\mathbf{h}}_i) = \hat{\mathbf{h}}_{i-1} \tag{2.7}$$

where $\hat{\mathbf{h}}_{i-1}$ represents the estimated target at layer $i - 1$. Let us assume that such inversion nets were trained properly to realize $g_i(.)$ for $i = \ell, ..., 1$. Then, layer specific targets could be defined according to the following recursive procedure.
Basis of recursion: first of all, if the layer i is the output layer, i. e. $i = \ell$, then $\hat{\mathbf{h}}_i = \hat{\mathbf{y}}$ and $g_\ell(\hat{\mathbf{y}}) = \hat{\mathbf{h}}_{\ell-1}$.
Recursive step: the target outputs for subsequent layers $(\ell - 1, ..., 1)$ are obtained by applying $g_i(.)$ to the estimated targets available at the adjacent upper (i.e. i-th) layer.
The actual error driven training procedure for the inversion net proposed herein modifies this basic framework in the following manner. Given the generic layer i for which we want to learn the inversion function $g_i(.)$, let us define a layer-specific

dataset $\mathcal{D}_i = \{(\mathbf{x}'_{i,j}, \hat{\mathbf{y}}'_{i,j}) | \ j = 1, ..., N\}$ where, omitting the pattern specific index j for notational convenience, the generic input pattern is $\mathbf{x}'_i = (\hat{\mathbf{h}}_i, \mathbf{e}_i)$ given by the concatenation of the target value at layer i (either known, if $i = \ell$, or pre-computed from the upper layers if $i < \ell$) and the corresponding layer-specific signed error $\mathbf{e}_i = \hat{\mathbf{h}}_i - \mathbf{h}_i$. Herein, $\mathbf{h}_i$ is the actual state of layer i of *dnet* upon forward propagation of its input, such that $\mathbf{x}'_i \in \mathbb{R}^{2 \times d_i}$. In turn, $\hat{\mathbf{y}}'_i$ is defined to be the state of the $(i-1)$-th layer of *dnet*, namely $\hat{\mathbf{y}}'_i = \mathbf{h}_{i-1}$.
Once the supervised dataset $\mathcal{D}_i$ has been built this way, the inversion net can be trained using standard BP with an early-stopping criterion. We say that this scheme is error-driven, meaning that the inversion net learns a target-estimation mapping which relies on the knowledge of the errors $\mathbf{e}_i$ stemming from the forward-propagation process in *dnet*. Once training of the inversion net is completed, the proper target-propagation step (from layer i to $i-1$) can be accomplished as follows. The inversion network is fed with the vector $(\hat{\mathbf{h}}_i, \mathbf{e}_i)$ where we let $\mathbf{e}_i = \mathbf{0}$ in order to get $g_i(\hat{\mathbf{h}}_i) = \hat{\mathbf{h}}_{i-1} \simeq f_i^{-1}(\hat{\mathbf{h}}_i)$. In so doing, the inversion net generates layer specific target that, once propagated forward by dnet, are expected to result in a null error, as sought. The resulting training procedure is formalized in Algorithms 1 and 2 in the form of pseudo-code. The algorithms assume the availability of two procedures, namely: $feedForward(net, \mathbf{x})$, realizing the forward propagation of an input pattern $\mathbf{x}$ through a generic neural network *net*; and $backpropagation(net, \mathcal{D})$, that implements the training of the network *net* via BP from the generic supervised training set $\mathcal{D}$.

In order to reduce the bias intrinsic to the training algorithm, target propagation is accomplished relying on a modified strategy, as in different target propagation scheme [39], accounting for the bias that the layer specific inversion nets $g_i(.)$ are likely to introduce in estimating the corresponding target outputs $\hat{\mathbf{h}}_{i-1}$. To this end we let

$$\hat{\mathbf{h}}_{i-1} = \mathbf{h}_{i-1} + g_i(\hat{\mathbf{h}}_i, \mathbf{0}) - g_i(\mathbf{h}_i, \mathbf{0}) \tag{2.8}$$

The rationale behind this equation is the following. First of all, $g_i(.)$ can be also applied to invert the actual state $\mathbf{h}_i$ of *dnet* instead of the target state $\hat{\mathbf{h}}_i$. Ideally, if the mapping realized by the inversion net were perfect, we would have $g_i(\mathbf{h}_i, 0) = \mathbf{h}_{i-1}$. To the contrary, since $g_i(.)$ is the noisy outcome of an empirical learning procedure, in practise $g_i(\mathbf{h}_i, 0) \neq \mathbf{h}_{i-1}$ holds, i.e. an offset is observed whose magnitude is given by $|g_i(\mathbf{h}_i, \mathbf{0}) - \mathbf{h}_{i-1}|$. Equation (2.8) exploits this offset as a bias corrector when applying $g_i(.)$ to the computation of $\hat{\mathbf{h}}_{i-1}$ as well. Note that whenever $g_i(\mathbf{h}_i, \mathbf{0}) = \mathbf{h}_{i-1}$ (unbiased inversion net) then the equation reduces to $\hat{\mathbf{h}}_{i-1} = g_i(\hat{\mathbf{h}}_i, 0)$, as before. The details of the present bias-correction strategy are handed out in [39].

Algorithm 1 Training of the inversion net

Input: initialized inversion net $invNet_i$ with $2 \times d_i$ input units and d_{i-1} output units, deep network *dnet*, training set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N\}$, layer i, targets $\hat{\mathbf{h}}_i$

Output: the trained inversion net $invNet_i$ for layer i, capable of computing $\hat{\mathbf{h}}_{i-1}$ from $\hat{\mathbf{h}}_i$

```
1: D_i = ∅
2: for j = 1 to N do
3:     feedForward(dnet, x_j)
4:     e_{i,j} ← ĥ_{i,j} − h_{i,j}
5:     x'_{i,j} ← (ĥ_{i,j}, e_{i,j})
6:     ŷ'_{i,j} ← h_{i−1,j}
7:     D_i = D_i ∪ {(x'_{i,j}, ŷ'_{i,j})}
8: end for
9: invNet_i = backpropagation(invNet_i, D_i)
```

Algorithm 2 Target propagation

Input: trained inversion net $invNet_i$, layer i, number of patterns k, targets to be propagated $\hat{\mathbf{h}}_{i,j}, j = 1, ..., N$

Output: the propagated targets $\hat{\mathbf{h}}_{i-1,1}, ..., \hat{\mathbf{h}}_{i-1,N}$

```
1: for j = 1 to N do
2:     e_{i,j} = 0
3:     x'_{i,j} = (ĥ_{i,j}, e_{i,j})
4:     ĥ_{i−1,j} = feedForward(invNet_i, x'_{i,j})
5: end for
```

Refinement of Deep Network Learning

The notions introduced in previous paragraphs and the pseudo-code contained in algorithms 1 and 2 clearly explain how a target propagation technique can be realized using the *error driven* approach. An attempt to train neural networks exclusively relying on this error driven target propagation algorithm (EDTP), without using the standard BP, has been done. It acts as in the following. After a standard neural network initialization step, starting from the topmost layer ℓ, for each layer i ($i = \ell, ..., 2$), targets are generated for the subsequent layer $i - 1$. Using this generated targets, a layer specific loss $\mathcal{L}_i(.)$ is defined. Now, from the bottommost layer to the topmost, layer weights are updated in order to minimize the associated $\mathcal{L}_i(.)$. Even if the results are promising, actually this pure target propagation training strategy does not overcome the standard BP algorithm. For such a reason, this target prop technique (algorithm 1 and 2) has been chosen to be the building block for a refinement technique for pre-trained neural network. The goal is to further improve the perfor-

mances of pre-trained networks, by keep on updating the network weights relying on the layer-specific loss, generated as before. This is expected to lead to improved performances especially in deep network, where lower layer weights suffer for the vanishing gradient problem [23, 6]. Therefore, it is well known that with standard BP the magnitude of the gradient reaching lower layers can be very small, and those weights are only weakly updated. On the other hand, updating weights using layer-specific losses do get rid of this issue, improving the network performances. The overall approach goes as follows:

1. The deep network is pre-trained via BP, as usual
2. Targets are propagated downward through the layers, as in Alg. 1 and 2
3. The network is trained layer-wise accordingly. This phase is said *refinement*

Algorithm 3 provides a detailed description of this refinement strategy. The algorithm invokes a routine `Initialize_Network`(*net*) used to randomly initialize a generic feed-forward neural network *net* before the beginning of the standard training phase. Finally, the routine `layer_backprop`($\mathbf{h}_{i-1,j}, \hat{\mathbf{h}}_{i,j}$) realizes the adaptation of the weights between layers $i-1$ and i (for $i = 1, ..., \ell$) via online gradient-descent. This application of gradient-descent uses $\mathbf{h}_{i-1,j}$ as input and $\hat{\mathbf{h}}_{i,j}$ as the corresponding target output. It is seen that extensions of the procedure to batch gradient-descent and/or multi-epochs training are straightforward by working out the skeleton of pseudo-code offered by Algorithm 3.

Algorithm 3 Refinement of network learning based on target propagation

Input: deep network *dnet*, supervised training set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^{k}\}$
Output: the refined network *dnet*
Procedure `network_refinement`$(dnet, D)$

1: **for** $j = 1$ to N **do** ▷ Layer state calculation
2: **for** $i = \ell$ to 1 **do**
3: **if** $i = \ell$ **then**
4: $\hat{\mathbf{h}}_i = \hat{\mathbf{y}}$
5: **end if**
6: $\mathbf{h}_i = F_i(\mathbf{x})$
7: $\mathbf{h}_{i-1} = F_{i-1}(\mathbf{x})$
8: **end for**
9: **end for**
10: **for** $i = \ell$ to 2 **do** ▷ Target propagation
11: `Initialize_Network`$(invNet_i)$
12: $invNet_i$ = `train_inv_net`$(invNet_i, dnet, \mathcal{D}, i, \hat{\mathrm{h}}_i)$
13: $\hat{\mathbf{h}}_{i-1,1}, ..., \hat{\mathbf{h}}_{i-1,k}$ = `target_prop`$(invNet_i, i, \hat{\mathbf{h}}_{i,1}, ..., \hat{\mathbf{h}}_{i,k})$
14: **end for**
15: **for** $j = 1$ to N **do** ▷ Layer-wise training
16: **for** $i = 1$ to ℓ **do**
17: **if** $i = 1$ **then**
18: $\mathbf{h}_{i-1,j} = \mathbf{x}_j$
19: **end if**
20: `layer_backprop`$(\mathbf{h}_{i-1,j}, \hat{\mathbf{h}}_{i,j})$
21: **end for**
22: **end for**

The algorithm so defined results being a promising refinement technique. It is especially indicated when dealing with deep multilayer networks having non-linear activation function, where the refinement of lower layers can imply a meaningful performance improvement, as sought.

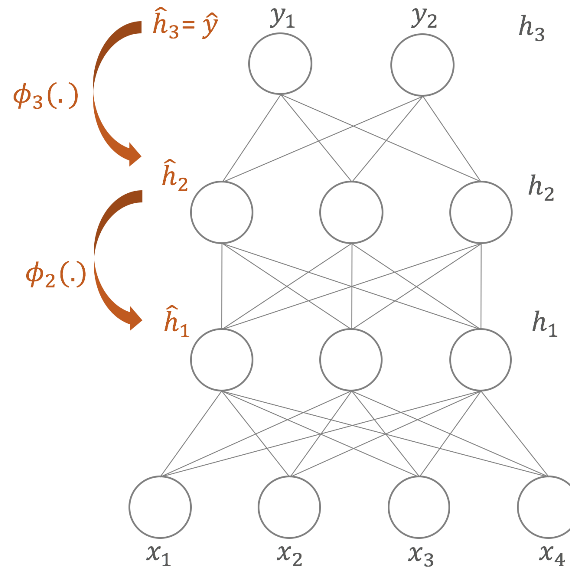

Figure 2.1: An illustration of how targets propagate downward through the network.

2.4 Residual Driven Target Propagation

In the previous sections we discussed at some length about the concept of target propagation. We introduced a technique, said "error driven target propagation" and explained how it can be used as an alternative technique to train neural networks without back propagating errors downward through the layers, as in standard BP. We then showed how this technique can be used as a refinement tool to further improve the performances of pre-trained networks. In this section we introduce another target propagation algorithm, said *residual driven target propagation* (RDTP). The motivation leading to the conceivement of this new approach is the following. Assuming to have a neural network *dnet* with a single hidden layer of size d, single output unit and an input layer of size n, and that this network was pre-trained using a supervised training-set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N\}$. As defined before, given a network having ℓ layers, target propagation consists in estimating the target values for layer $\ell - 1$, given the targets at layer ℓ, by means of a function $\phi(.)$ such that $\hat{\mathbf{y}}_{\ell-1} = \phi(\hat{\mathbf{y}}_\ell)$, and such that $f_\ell(\phi(\hat{\mathbf{y}}_\ell)) \simeq \hat{\mathbf{y}}_\ell$, where $f_\ell(.)$ is the function realized by the ℓ-th layer (using the definitions introduced in section 1.1). As we already know, $\phi(.)$ is a layer-specific function. Here, instead of estimating the targets $\hat{\mathbf{h}}_{\ell-1}$ as in the usual way, we aim at estimating the residual values $\mathbf{z}_{\ell-1}$, where with the term "residual" we mean the difference between the actual state $\mathbf{h}_{\ell-1}$ and the desired, although unknown, target value $\hat{\mathbf{h}}_{\ell-1}$, such that $\mathbf{h}_{\ell-1} + \mathbf{z}_{\ell-1} = \hat{\mathbf{h}}_{\ell-1}$. This approach is advantageous mostly when dealing with pre-trained model, for the following considerations. Assuming to train a neural network using $\mathcal{D}$, and that this trained model reaches top

performance (as 100% accuracy in a classification task or zero mean squared error in a regression task), here we have that $\mathbf{h}_i = \hat{\mathbf{h}}_i$ for $i = 1, ..., \ell$, that is that layers state values exactly correspond to layer target values, because we cannot do better, and these values are just what was needed. In this situation, it is clear that the residual value $\mathbf{z}_{\ell-1} = \hat{\mathbf{h}}_{\ell-1} - \mathbf{h}_{\ell-1}$ is null at the end of the training process; while starting from the first epoch, it kept decreasing, until being zero. From the last analysis, indicating with t the particular training epoch, and with τ a certain time period, we can say that $\mathbf{z}_{\ell-1}(t+\tau) \leq \mathbf{z}_{\ell-1}(t)$ if the training process is such that the loss function keeps decreasing at the same time. From the afore considerations, it is easy to understand that after a pre-training process we have that $|\mathbf{z}_{\ell-1}| \ll |\mathbf{h}_{\ell-1}|$ with a consequent more bounded co-domain for the inversion function $\phi(.)$. This basically translates in a more error-robust target propagation technique.
There is another motivation that makes residual driven target propagation even more interesting. To the contrary of what happens with other TP algorithms, in RDTP, if for a particular input pattern does not stand any output error, the residual estimated is just zero (as will be shown later), $\mathbf{z}_{\ell-1} = \mathbf{0}$, and $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{0}$, meaning that the actual state is just the target. This guarantees that, in case of a layerwise training phase, the knowledge contained into the network will be kept safe, in other words, what has been correctly learned from the network in a pre-training phase will not be altered. This is not guaranteed with other target prop techniques.
RDTP works as described in the following lines. The core of the present approach is the estimation of the residues $\mathbf{z}_{\ell-1}$ at layer $\ell-1$, given the network output error $(\hat{\mathbf{y}} - \mathbf{y})^2$. Once the residues are estimated, we can easily define the target values for layer $\ell-1$ as $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{z}_{\ell-1}$, where we have omitted the dependence on the input pattern. Given *dnet*, as defined at the beginning of the section (where ℓ and $\ell-1$ in this case correspond to the output and hidden layer of the MLP respectively), using an ad-hoc notation (indicating with the apexes and subscripts the layer-specific and the neuron-specific indexes respectively), bearing in mind that we have assumed an output layer with a single unit, we write the network output as

$$y = \sigma_\ell \left(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} h_u^{(\ell-1)} + b^{(\ell)} \right) \tag{2.9}$$

with

$$h_u^{(\ell-1)} = \sigma_{\ell-1} \left(\sum_{k=1}^{n} w_{u,k}^{(\ell-1)} x_k + b_u^{(\ell-1)} \right) \tag{2.10}$$

where the input $\mathbf{x} \in \mathbb{R}^n$. Now we can say that the generic network target value is equal to

$$\hat{y} = \sigma_\ell \left(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} \hat{h}_u^{(\ell-1)} + b^{(\ell)} \right) \tag{2.11}$$

where $\hat{h}_u^{(\ell-1)}$ for $u = 1, ..., d_{\ell-1}$ are the hidden layer target values. Knowing that $z_u^{(\ell-1)} = \hat{h}_u^{(\ell-1)} - h_u^{(\ell-1)}$, and consequently $\hat{h}_u^{(\ell-1)} = h_u^{(\ell-1)} + z_u^{(\ell-1)}$, we can rewrite the latter as

$$\hat{y} = \sigma_\ell\Big(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)}(h_u^{(\ell-1)} + z_u^{(\ell-1)}) + b^{(\ell)}\Big) \tag{2.12}$$

$$= \sigma_\ell\Big(\underbrace{\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} h_u^{(\ell-1)}}_{\tilde{a}} + \underbrace{\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}}_{\tilde{a}_z} + b^{(\ell)}\Big) \tag{2.13}$$

and in order to derive the residual values, we need to explicit $\tilde{a}_z$. After these considerations, we can also define the target output as

$$\hat{y} = y + y_z \tag{2.14}$$

where y_z is the output component related to the residues, that we consider such that $y_z = \sigma_\ell(\tilde{a}_z)$. From which we can do

$$y_z = \hat{y} - y \tag{2.15}$$

$$\tilde{a}_z = \sigma_\ell^{-1}(y_z) \tag{2.16}$$

Now we can estimate the internal residues starting from $\tilde{a}_z$. Where we know that $\tilde{a}_z = \sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}$. The idea behind the estimation of the z_u values is the following. We state that when a pattern $\mathbf{x} \in \mathcal{D}$ is fed into the network *dnet*, and an output error y_z is detected, each unit at layer $\ell - 1$ is responsible in somehow for that particular error. We formalize the concept of "responsibility for the u-th neuron" of a generic layer i of size d_i as a quantity $r_u^{(i)}(\mathbf{x}) \in [0, 1]$ such that

$$\sum_{u=1}^{d_i} r_u^{(i)}(\mathbf{x}) = 1 \tag{2.17}$$

meaning that the outcomes of all units in layer i in response to the particular input $\mathbf{x}$, entail a notion of responsibility, which is distributed among all the neurons of the layer. We defined the responsibility of a certain neuron u on a given pattern $\mathbf{x}$ as the outcome of the particular neuron, normalized over the outcomes of all the neurons within layer:

$$r_u^{(i)}(\mathbf{x}) = \frac{\sigma_i(a_u)}{\sum_{k=1}^{d_i} \sigma_i(a_k)} \tag{2.18}$$

where a_u and a_k are the activations of the generic units u and k. We will optionally omit the dependence on $\mathbf{x}$ for simplicity. The last step, who leads to the estimation of the single-units residues $z_u^{(i)}$, $u = 1, ..., d_i$, is based on the assumption that the

higher is the responsibility on a certain error, the higher is the "needed" residue $z_u^{(i)}$. This arises from the idea that the outcome of a generic unit u is proportional to the competence that it has on the given input pattern, or more formally on the region of the domain that the pattern belongs to. If the outcome of the neuron is close to zero, with a consequent null responsibility value, $r_u^{(i)}(\mathbf{x}) \simeq 0$, it means that u it is not competent on $\mathbf{x}$. This entails a intrinsic sparsity property in neural networks, where only a fraction of neurons are active at the same time. In order to keep this property unchanged (in case of a further layer-wise training), we define residues proportionally to the responsibilities of the single neurons. In particular knowing that $\tilde{a}_z = \sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}$, we empirically decompose $\tilde{a}_z$ assuming that each component of the sum is proportional to the corresponding responsibility value as $r_u^{(\ell-1)} \tilde{a}_z = w_u^{(\ell)} z_u^{(\ell-1)}$, from which we have

$$z_u^{(\ell-1)} = \frac{r_u^{(\ell-1)} \tilde{a}_z}{w_u^{(\ell)}} \tag{2.19}$$

The whole procedure is described in Algorithm 4, where we have also introduced a pattern-specific index j for each parameter.

Algorithm 4 Residual Driven Target Propagation Algorithm

Input: training set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N\}$, the network *dnet*, the output layer $i+1$
Output: the propagated targets at layer i. For *dnet*, layer i corresponds to the single hidden layer.
Procedure `residual_target_prop`$(dnet, \mathcal{D}, i+1)$

1: **for** $j = 1$ to N **do**
2: $\quad y_j = F_{i+1}(\mathbf{x}_j)$
3: $\quad y_{z,j} = \hat{y}_j - y_j$
4: $\quad \tilde{a}_{z,j} = \sigma_{i+1}^{-1}(y_{z,j})$
5: $\quad$ **for** $u = 1$ to d_i **do**
6: $\qquad r_u^{(i)}(\mathbf{x}_j) = \frac{\sigma_i(a_u)}{\sum_{s=1}^d \sigma_i(a_s)}$
7: $\qquad z_u^{(i)} = \frac{r_u^{(i)} \tilde{a}_z}{w_u^{(i+1)}}$
8: $\qquad \hat{h}_u^{(i)} = h_u^{(i)} + z_u^{(i)}$ $\quad \triangleright\ h_u^{(i)} = \sigma_i(a_u)$
9: $\quad$ **end for**
10: $\quad \hat{\mathbf{h}}_{i,j} = (\hat{h}_1^{(i)}, ..., \hat{h}_{d_i}^{(i)})$
11: **end for**

2.5 Gradient Based - Residual Driven Target Prop

The RDTP algorithm described in the previous section does not require auxiliary neural networks for the estimation of targets, as it happen in other techniques, making this a much faster alternative for target propagation (from a computational perspective), because it consists in a well define set of operations that have to done only once for each single pattern . Anyway this approach has a weak point. It can be applied only to neural networks having single output unit. The immediate consequence is that, it is not possible to further propagate targets from layer $\ell - 1$ to layer $\ell - 2$, because $\ell - 1$ is in most of the cases is a layer of size greater than one. To the state of the art, RDTP as it is, can only be applied to standard MLPs with single output units. Therefore, to overcome this limitations, an extended version of the algorithm has been conceived. This is called *Gradient Based RDTP*. The main idea is the same of the standard RDTP, that is to estimate residual values $\mathbf{z}_{\ell-1}$ for layer $\ell - 1$ such that we can derive the propagated targets as $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{r}_{\ell-1} \odot \mathbf{z}_{\ell-1}$, where the residues are multiplied element-wise by the responsibility of neurons (we have used a vectorial notation) in order to keep unchanged the sparsity property of the neural network, as mentioned before. Here we estimate the residues $\mathbf{z}_{\ell-1}$ with a gradient descent approach, defining the following

$$\hat{\mathbf{h}}'_\ell = \sigma_\ell\Big(W_\ell(\mathbf{h}_{\ell-1} + \mathbf{r}_{\ell-1} \odot \mathbf{z}_{\ell-1}) + \mathbf{b}_\ell\Big) \tag{2.20}$$

and minimizing the loss function $\mathcal{L}(\hat{\mathbf{h}}_\ell, \hat{\mathbf{h}}'_\ell) = \|\hat{\mathbf{h}}_\ell - \hat{\mathbf{h}}'_\ell\|_2^2$, iteratively updating $\mathbf{z}_{\ell-1}$ as

$$\mathbf{z}'_{\ell-1} = \mathbf{z}_{\ell-1} - \eta \frac{\partial \mathcal{L}(\hat{\mathbf{h}}_\ell, \hat{\mathbf{h}}'_\ell)}{\mathbf{z}_{\ell-1}} \tag{2.21}$$

that is a typical gradient descend update rule. It can be done in online or batch style. In order to propagate the targets to the preceding layers, the process must be repeated changing the layer-specific parameters. In algorithm 5 it is shown the detailed process, where $calculate_layer_resp(net, i, \mathbf{x})$ calculates the responsibilities for layer i of network net element-wise as in eq. (2.18), in response to the input $\mathbf{x}$, while $estimate_residues(net, \mathbf{x}, \mathbf{r}_i, \hat{\mathbf{h}}_{i+1}, \mathbf{h}_i)$ performs the residues estimation for layer i using the update rule in eq. (2.21) until a stopping criterion has met; furthermore a pattern specific index has been introduced for all involved parameters.

Algorithm 5 Gradient Based - Residual Driven Target Prop Algorithm

Input: training set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N\}$, the network *dnet*, the layer i
Output: the propagated targets at layer $i-1$
Procedure `gradient_based_RDTP`$(dnet, \mathcal{D}, i)$

1: **for** $j = 1$ to N **do**
2:     **if** $i = \ell$ **then**
3:         $\hat{\mathbf{h}}_{i,j} = \hat{\mathbf{y}}_j$
4:     **end if**
5:     $\mathbf{h}_{i-1,j} = F_{i-1}(\mathbf{x}_j)$
6:     $\mathbf{r}_{i-1,j}$ = `calculate_layer_resp`$(dnet, i-1, \mathbf{x}_j)$
7:     $\mathbf{z}_{i-1,j}$ = `estimate_residues`$(dnet, \mathbf{x}_j, \mathbf{r}_{i-1}, \hat{\mathbf{h}}_{i,j}, \mathbf{h}_{i-1,j})$
8:     $\hat{\mathbf{h}}_{i-1,j} = \mathbf{h}_{i-1,j} + \mathbf{r}_{i-1,j} \odot \mathbf{z}_{i-1,j}$
9: **end for**

This herein introduced technique, solves some of the limitation of RDTP. Anyway it is of immediate understanding the the estimation of the residues scales linearly with the number of patterns of the training set. So attention has to be paid to this aspect.

2.6 Experiments with the Refinement Algorithm

In this section we show the performances of the *refinement* algorithm introduced in the section 2.3 and widely described in [36] . Experiments were conducted on the popular MNIST dataset [38], 70000 patterns representing pixel-based images of handwritten digits (10 classes overall) having dimensionality of 784. A 10-fold crossvalidation strategy was applied, where for each fold 80% of the data were used for training, 10% for validation, and 10% for test. Our aim here is to exploit MNIST as a significant and difficult learning task suitable to assess the effectiveness of the present approach, and to compare the proposed algorithms to established non- convolutional feed-forward networks and target propagation methods previously applied to MNIST as [51, 39]. Gradient-based training of the main network *dnet* (the classifier) relied on RMSProp [53], and learning rate of 0.01, while for the inversion net and the layer-wise refinement of *dnet* the Adam [32] variant was used, with learning rate of 0.001. The network hidden layers were composed of 140, 120 and 100 neurons respectively, with sigmoid activation function. All components were trained using MSE loss. The inversion net hidden layer was composed of 200 units with sigmoid activation functions.

Table 2.1: Accuracy on MNIST 10-fold classification task (avg. $\pm$ std. dev. on a 10-fold crossvalidation)

Algorithm	Training	Test
RMSProp	99.48 ± 0.13	$98,12 \pm 0.05$
Target Propagation	87.30 ± 0.29	86.64 ± 0.27
Refinement	99.65 ± 0.08	98.27 ± 0.06

Table 2.2: Comparison between the proposed algorithm and the established approaches, in terms of error rate and number of adaptive parameters.

Algorithm	Test Error	#Parameters
Refinement	1.73 ± 0.06	3.04×10^5
[39]	1.94	5.36×10^5
[51]	1.6	1.28×10^6

Table 2.1 compares the accuracies for *dnet* trained with RMSProp, bare target propagation and with the refinement. In terms of learning capabilities (evaluated

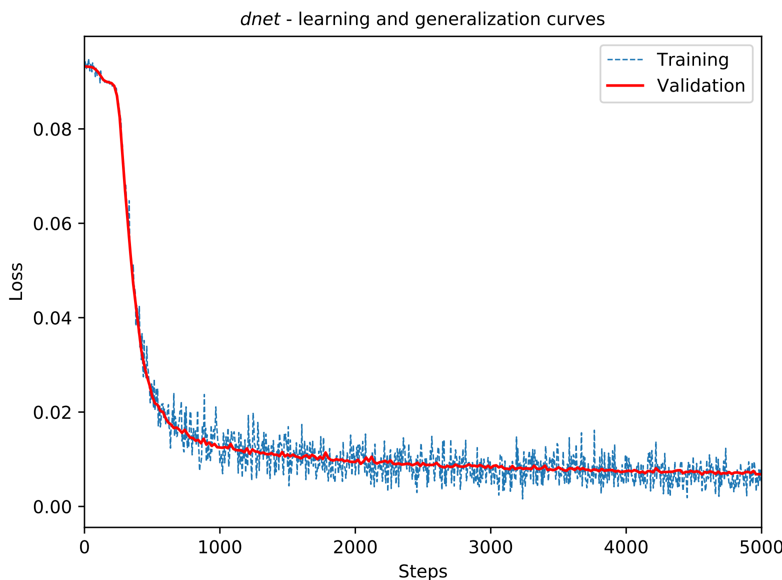

Figure 2.2: Learning and generalization curves for *dnet*.

on the training set), the refinement (that consists in applying target propagation to the pretrained *dnet*) yelds a 32.75% average error rate reduction over RMSProp. In terms of generalization (evaluated on the validation set) an average error rate reduction of 8.20% was observed. Table 2.2 offers a comparison among MNIST classifiers based on non-convolutional feedforward neural networks using no augmentation of the training set. The comparison involves the error rate as observed and the number of parameters of the model, that is an index of model complexity. It is seen that the error rate achieved by the proposed refinement algorithm is in the middle between its competitors, but the complexity of the machine is dramatically smaller. Figure 2.2 presents the learning and generalization curves (mean squared error on training and validation sets, respectively) obtained running regular BP learning of *dnet* in one of the 10-folds of the present experiment. Note that the loss used to plot the learning curve was evaluated, from step to step, on the corresponding training mini-batch only, while the generalization curve was always evaluated on the whole validation set. In fig. 2.3 are the learning and generalization curves of the layer-specific gradient-descent adaptation of the weights in the 1st, 2nd, and 3rd hidden layers of *dnet*, respectively, by means of the application of the procedure layer $backprop(.)$ to the target propagated via the inversion net. Figure 2.4 shows the curves for $layer_backprop(.)$ applied to the weights in the topmost layer of dnet. Although eventually one is interested in solving the original learning problem, it is seen that the layer-specific sub-problems are actually difficult high-dimensional learning problems, which may just not admit any sound single-layered solution. This explains the observed difficulties net by gradient descent in minimizing the

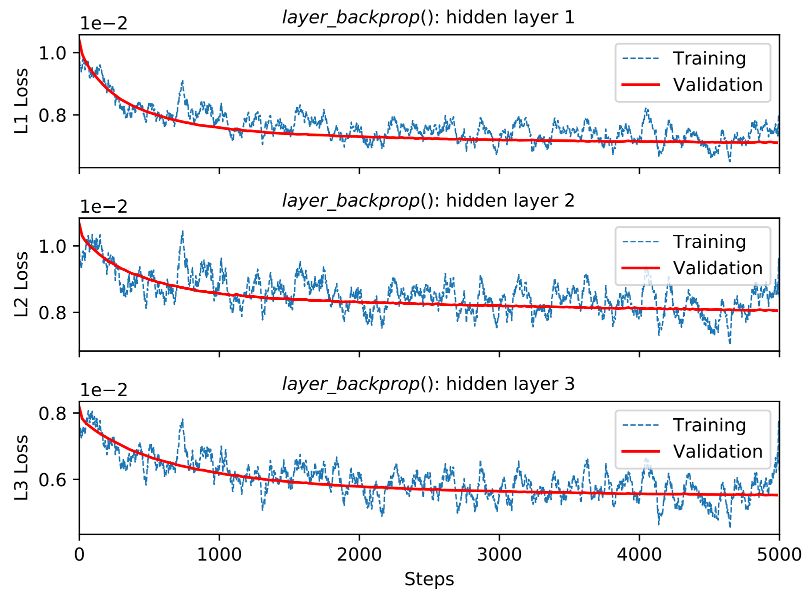

Figure 2.3: Learning and generalization curves of the procedure *layer_backprop*(.) applied to the three hidden layers of *dnet*.

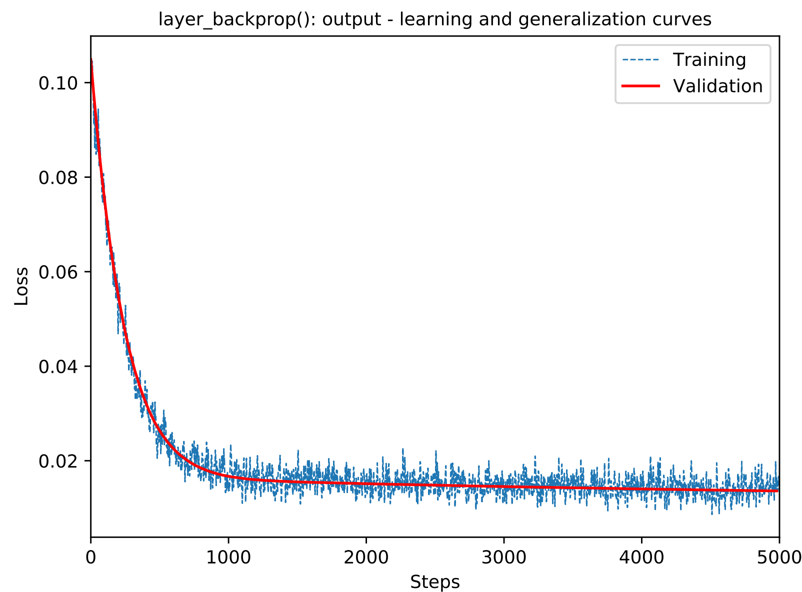

Figure 2.4: Learning and generalization curves of the procedure *layer_backprop*(.) applied to the output layer of *dnet*.

corresponding layer-specific loss functions.
Target propagation emerges as a viable approach to learning and refinement of deep neural networks, tackling the vanishing-gradient issues stemming form application of plain BP to deep architectures. Albeit preliminary, the empirical evidence stresses that the proposed refinement strategy yields classification accuracies that are in line with the state- of-the-art algorithms for training feed-forward networks.

2.7 Remarks

Propagation of targets outputs to internal layers of neural networks is nowadays an under investigated research area. In this chapter an overview of most interesting works done in this direction has been made, especially of those most related to this thesis. Three new algorithms, *Error Driven Target Prop* (EDTP), *Residual Driven Target Prop* (RDTP) and *Gradient Based* RDTP (GB-RDTP) conceived during the development of this thesis have been introduced. While all these techniques share the same goal, the estimation of target for internal layers, they are based on different ideas and approaches. The EDTP, while resulting slower than the RDTP due to its neural-based nature, it can be used in all possible situations/architectures. On the contrary, the RDTP results in a much faster technique, but usable only when dealing with MLPs having single output unit. The same considerations done for EDTP are valid also for GB-RDTP, because the latter uses a gradient descent technique for the estimation of residues, and does not suffer of any architectural limitations. In next chapters we will see how the contemporary use of more of these techniques became very useful for the development of the present work.

Chapter 3

Depth Growing Neural Networks

Managing DNNs and relative learning strategy can be very challenging. Knowing the suitability of a certain DNN architecture for a given learning task $\mathcal{T}$ requires at least the expectation of a certain number of training epochs. At the end of this process we are able to measure performances of the model, and establish if the chosen architecture fits or if a new training process with a different architecture is needed. This described process, especially if dealing with lot of data and big architectures, can result in lot of time wasting.

In last years the community is addressing efforts in the direction of always more automated techniques to create and train DNN models. In this context, the idea of the *depth growing neural network* model (DGNN) comes out. Basically it was conceived with the aim of realizing a new neural network model, able to overcome some of the aforementioned issues, and that would be easier to train.

It mainly deal with the topic of neural architecture search: size and number of layers, and type of activation functions. The key idea that drove the conception of the DGNN model, was to define a neural model whose performance improvement was due to a progressive adaptation of its activation functions to the problem at hand. The aim was to have models whose outcomes were resulting from linear (and non-linear) combination of more (or very) complex non-linearities. This idea has an important role in the architectural imprint of the DGNN model.

3.1 Related Works (adaptive activation functions)

In this section we discuss few affine models, that are able to autonomously set some of their architectural configurations. Here we focus on the topic of autonomously finding the best activation functions setting for a given network, that can be considered as a sub-task of the more general one of finding the right neural network architecture. In particular we analyze models aimed at learning their activation functions.

One of the first works in the area of learnable activation functions is the work [11], where the author tries to learn the *smoothness* θ of a sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x/\theta}}$. One same line is the work of [54], where the author further generalizes the sigmoid activation function as $\sigma(x) = \frac{\lambda}{1+e^{-x/\theta}}$ and aims at learning the amplitude λ.

An important work in the area of learnable activation function is [26], the maxout network. This consists in a feedforward network having special kind of neurons said maxout units. These units i have an activation function of the form $y_i = \max(xW_1 + b_1, xW_2 + b_2, ..., xW_k + b_k)$ where x is the previous layer state and $W_1, b_1, ..., W_k, b_k$ are learnable parameters. If we set all components but one to zero, it is easy to see that in this way we obtain the standard ReLU [24] activation function. By learning the appropriate parameters we could potentially approximate any kind of activation function, at the cost of a substantial increase of the number of parameters to learn.

Similar is the work of [1], where the adaptive piecewise linear unit (APL) is defined. It consists in neurons with the following activation function: $y_i(x) = \max(0, x) + \sum_{s=1}^{S} a_i^s \max\{0, -x + b_i^s\}$, where S is a fixed hyperparameter and a_i^s and b_i^s are learnable parameters, a_i^s determines the slope and b_i^s the bias.

Among existing works on the topic of learning the activation functions, we need to mention the work of [10], that inspired the idea of DGNN model. Here the author aims at learning the activation functions of the hidden layer of an MLP using auxiliary smaller neural networks, whose training set was generated using a gradient based approach. It is similar to the work of this Thesis, with the difference that the DGNN model can recursively replace its internal neurons, aiming in this way at defining a deep architectures. So we can say that while the goal of [10] was limited to the search of the best fitting activation functions, here we aim at finding the best architecture. Furthermore, the research activity on the DGNN model lead to the development of new target propagation techniques, and some of them are used in the DGNN training process.

3.2 The DGNN Model

We define the depth growing neural network model, said DGNN for convenience, as a neural network having a single internal (hidden) layer composed of particular neurons said *meta-neurons*, defined here; while input and output layers have standard neurons. The particularity of meta-neurons is that each of them, during the training process can eventually be replaced with a neural network, having single internal layer, that we call *subnet*, whose architecture will be specified later. The consequence of this replacement is that the activation function realized by the specific meta-neuron, is now realized by a subnet, that regardless of its architecture, is surely able to realize more complex functions. Indeed, among the goals of the framework is that activation functions of meta-neurons must be learned by their replaced subnets. This process of replacing the meta-neuron with a subnet is said *growing step*, and we say that the meta-neuron *evolves* in a neural network. While the idea of realizing activation function with small neural network is not new [10], one of the aspects that make this framework unique, is that subnet's single hidden layer in turn is composed of meta-neurons, realizing in this way a potential recursive growing process, with a consequent definition of a deep multilayer network. This strategy is aimed at realizing a model with a greater computational power w.r.t. MLPs, that can be compared with standard deep neural networks. The interesting novelty, that can make this framework appealing, is that this growing process may be seen as a natural evolution of the network internal architecture structure, leading to a configuration that best fits the computational needs for the given learning problem.

In the following lines we list the components involved in the framework, in order to give a more formal definition. Given a DGNN model that we call *net*, with its internal layer having size d_{net}, we have:

1. *Meta-neurons* q_i , with $i = 1, ..., d_{net}$
2. *Subnet* S_i, ($i = 1, ..., d_{net}$) is the network used to replace the meta-neuron q_i. S_i internal layer has size d_{S_i}.
3. $pa(S_i)$ is the parent network of S_i, whose particular meta-neuron q_i evolved in the subnet S_i.
4. $rel(S_i)$ is the set of all subnets S_j, ($j = 1, ..., d_{pa(S_i)}$) generated by the evolution of meta-neurons of $pa(S_i)$.
5. $depth(.)$ is a function that associate an integer value to a subnet or meta-neuron. Meta-neurons q_i ($i = 1, ..., d_{net}$) of *net* have $depth(net) = depth(q_i) = 0$ because they have not been generated by a growing process, so they have not a parent network. The subnet S_i (and its meta-neurons) generated by the evolution of

q_i $(i = 1, ..., d_{net})$ will have $depth(S_i) = 1$. In general we have $depth(Si) = depth(pa(Si)) + 1$. The depth of a subnet corresponds to the depth of its meta-neurons. The depth of a just initialized DGNN is zero.

6. $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N\}$ with $\mathbf{x}_j \in \mathbb{R}^n$, $\hat{\mathbf{y}}_j \in \mathbb{R}^m$ is the dataset used to train a certain DGNN.

7. $\mathcal{D}_i^{[k]}$ is the *subnet-specific* dataset aimed at training the subnet $S_i^{[k]}$. We indicate with apexes within the square brackets "$[k]$" the depth of the component (subnet or meta-neuron), so that $x^{[k]}$ and $y^{[k]}$ are the generic input and output of a subnet at depth k, and we write

$$\mathcal{D}_i^{[k]} = \{(x_{ij}^{[k]}, \hat{y}_{ij}^{[k]})_{j=1}^N)\},\ x_{ij}^{[k]} \in \mathbb{R},\ \hat{y}_{ij}^{[k]} \in \mathbb{R} \tag{3.1}$$

with this convention we say that $\mathbf{x}^{[0]}$ and $\hat{\mathbf{y}}^{[0]}$ are the input and output of a subnet at depth 0, that corresponds to the just initialized DGNN, and $\mathbf{x}^{[0]} = \mathbf{x}$ and $\hat{\mathbf{y}}^{[0]} = \hat{\mathbf{y}}$, therefore $\mathcal{D}^{[0]} = \mathcal{D}$. In general when the depth is not indicate, it is assumed to be zero. The depth indication is especially needed when illustrating recursive procedures; it is optionally being replaced by an accent " $'$ ", when indicating that the subnets/neurons have *depth* = 1.

Here we give a formalization of the mapping operations defined in a DGNN.
In figure 3.1 is represented a newly initialized DGNN, having single internal layer. From now on, in order to rely on a depth-specific notation, we introduce the following. Knowing that the building blocks of this framework are subnets with single hidden layer, that regardless of the type of neurons correspond to the well known MLPs and that the same DGNN when just initialized is an MLP, we define the functions realized by these building blocks, as the composition of functions $f_L(.)$ and $f_U(.)$ realized by their lower and upper layers respectively. So in case of a depth-0 DGNN, the function $f(.)$ realized by the model can be represented as

$$\mathbf{y} = f(\mathbf{x}) = f_U(\sigma(f_L(\mathbf{x}))) \tag{3.2}$$

where $f_L(\mathbf{x}) = W_L\mathbf{x} + \mathbf{b}_L$ is a linear mapping, $\sigma(.)$ is an elementwise activation function and $f_U(\mathbf{o}) = \sigma(W_U\mathbf{o} + \mathbf{b}_U)$, where $\mathbf{o}$ is the outcome of the lower layer computation. Explaining the depth-specific notation, we can re-write the (3.2) as $\mathbf{y} = f^{[0]}(\mathbf{x}) = f_U^{[0]}(\sigma^{[0]}(f_L^{[0]}(\mathbf{x})))$, and in case of a depht-1 DGNN, where all the meta-neurons evolved in respective subnets, we have that the mapping is realized as

$$\mathbf{y} = f_U^{[0]}(f^{[1]}(f_L^{[0]}(\mathbf{x}))) \tag{3.3}$$

where $f^{[1]}(.)$ stands for the mapping realized by the depth-1 layer (set of subnets at depth-1), and therefore we can write

$$\mathbf{y} = f_U^{[0]}(f_U^{[1]}(\sigma^{[1]}(f_L^{[1]}(f_L^{[0]}(\mathbf{x}))))). \tag{3.4}$$

It is easy to note that what differs (3.2) from (3.3) is that in the last one the activations are now realized by $f^{[1]}(.)$.

So basically the outcome of a depth-k DGNN can be represented as the following composition of layer-specific mapping functions:

$$f_L^{[0]} \to f_L^{[1]} \to ... \to f_L^{[k]} \to \sigma^{[k]} \to f_U^{[k]} \to ... \to f_U^{[1]} \to f_U^{[0]} \tag{3.5}$$

that corresponds to

$$\mathbf{y} = (f_U^{[0]} \circ f_U^{[1]} \circ ... \circ f_U^{[k]} \circ \sigma^{[k]} \circ f_L^{[k]} \circ ... \circ f_L^{[1]} \circ f_L^{[0]})(\mathbf{x}) \tag{3.6}$$

Furthermore, to indicate the outcomes at a particular depth $i < k$, we define $F_L^{[i]}$ and $F_U^{[i]}$ as seen in the following schema

$$\underbrace{\overbrace{f_L^{[0]} \to ... \to f_L^{[i]}}^{F_L^{[i]}} \to ... \to f_L^{[k]} \to \sigma^{[k]} \to f_U^{[k]} \to ... \to f_U^{[i]}}_{F_U^{[i]}} \to ... \to f_U^{[0]} \tag{3.7}$$

From which we write

$$F_L^{[i]} = f_L^{[i]}(f_L^{[i-1]}...(f_L^{[0]}(\mathbf{x})) \tag{3.8}$$
$$F_U^{[i]} = f_U^{[i]}(f_U^{[i+1]}...(f_U^{[k]}(\sigma^{[k]}(F_L^{[k]}(\mathbf{x})))) \tag{3.9}$$

.

So far we gave the definition of the DGNN model, we described the philosophy behind the framework, we listed its main components and formalized its mapping dynamics. In the next section we describe the steps to take in order to train a DGNN model.

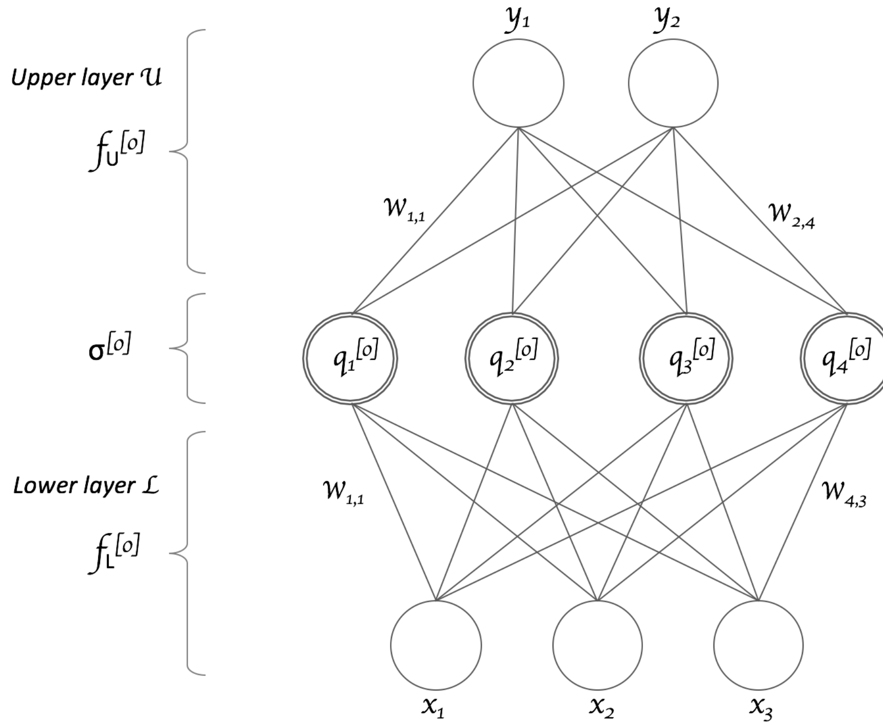

Figure 3.1: Representation of a DGNN having depth $k = 0$. The meta-neurons are represented with a custom circular shape. On the left side are indicated the mapping functions associated to the single layers.

3.3 Training Algorithm

From the description of this newly defined model held in the previous section, it is clear that DGNN has a modular structure: it is basically a deep neural network composed of smaller neural networks , that we call subnets, added during the learning process. As it is, it requires a particular learning strategy, that can be summarized in the following points.

1. Given $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^{N}\}$ with $\mathbf{x}_j \in \mathbb{R}^n$, $\hat{\mathbf{y}}_j \in \mathbb{R}^m$, define the architecture of the particular DGNN said *net*, such that input and output layers have size n and m respectively and choose the hidden layer size d_{net}.

2. Train the DGNN *net* using backpropagation as in the usual way. This is what we call the *first training phase*.

3. If the network performances are not satisfying, let the network *grow*, until some stopping criteria are met. This corresponds to the *second training phase*.

This sequence represents the basic steps to take in order to train a DGNN model. While the first and second steps are clear, because correspond to what happen in

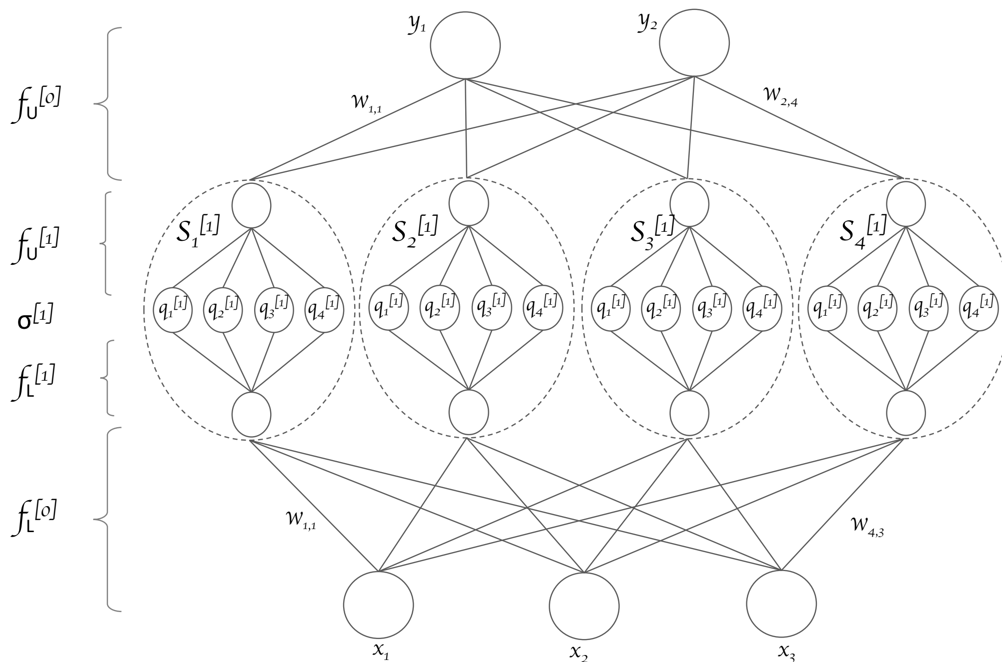

Figure 3.2: Representation of a DGNN having depth $k = 1$ (after a growing step). The meta-neurons at depth $k = 0$ have been replaced by subnets $S_1, ..., S_4$ having depth $k = 1$. Here all meta-neurons have been replaced, and all subnets have the same architecture.

classic DNN models, the third requires a much more in-depth discussion. Here we give a detailed description.

The Growing Step

The growing step is the backbone of the DGNN framework. It leads the DGNN from its initial condition, to a situation where it develops an its own configuration adapted to the needed computational power (in terms of units and layer). Given a partially trained DGNN, who performed the first training phase (depth k=0), assuming that we want the model to grow, the following steps must be followed.

1. **Selection of an appropriate architecture for the subnet S_i'**. The typical architecture (see fig. 3.2) consists in one neuron for input and output layer, and an arbitrary number d_{S_i} of meta-neurons for the internal layer (empirical process). Basically this step consists in choosing hidden layer size of the subnet.

2. **Generation of subnet-specific datasets**. The DGNN as it is, regardless of the type of neurons, consists in a neural network with single hidden layer. Thinking at the subnet-specific dataset $\mathcal{D}_i'$ for S_i', we have that $x_{i,1}', ..., x_{i,N}'$ are inputs to the particular meta-neuron q_i, while targets $\hat{y}_{i,1}', ..., \hat{y}_{i,N}'$ correspond to the values that we wanted q_i to generate. Knowing that a DGNN can be seen as a MLP

with special hidden neurons, the need of creating subnet-specific datasets for all subnets S'_i $(i = 1, ..., d_{net})$ acts as follows. Target values $\{(\hat{y}'_{i,j})_{j=1}^N\}$ are nothing but the propagation of output targets to the hidden layer (holding meta-neurons at depth-0). From a vector perspective, the generic propagated target can be seen as $\hat{\mathbf{y}}' = \phi_{TP}(\hat{\mathbf{y}})$, where $\phi_{TP}(.)$ is a generic target propagation function. Input values $\{(x'_{i,j})_{j=1}^N\}$ are generated feeding the network with input patterns and getting activations of the depth-0 layer, $\mathbf{x}' = F_L^{[0]}(\mathbf{x})$ (minding that lower layers have linear activation function). The propagation of target can be addressed using appropriate target propagation algorithms seen in the previous chapter.

3. **Choice of meta-neurons to replace**. Once we have the subnet-specific dataset for all meta-neurons of the layer, we can estimate an error measure for each meta-neuron, as $e_i = \sum_{j=1}^N (y'_{i,j} - \hat{y}'_{i,j})^2$, where $y'_{i,j}$ is the real outcome of the i-th meta-neuron with respect to the j-th pattern. This step may also be performed for all meta-neurons at the same time in a vector way. Based on this simple verification, a set $\mathcal{P}$ of bed-performing meta-neurons is chosen for the growing process.

4. **Meta-neurons replacement**. Each meta-neuron $q_i \in \mathcal{P}$ is replaced with the associate subnet S'_i.

5. **Training of the subnets**. The replacing subnets S'_i, $i = 1, ..., |\mathcal{P}|$, are trained with standard backpropagation algorithm as in the usual way, using the subnet-specific datasets $\mathcal{D}'_i$ ad-hoc generated.

The performing of the above sequence of steps, defines a single growing step for the DGNN model. As said since the beginning, the growth process can be recursive, so that a meta-neuron belonging to a certain subnet (generated by the evolution of another meta-neuron), can in turn grow and *evolve* in another subnet. Anyway, for each growing step, at whatever level of recursion it is, all the above defined steps must be always executed.

Subnet-specific Dataset Creation

As seen above, one of most important steps involved in the growing process is the generation of the subnet-specific datasets. It is directly related to selection of meta-neurons to grow and the training of replacing subnets. Once a subnet S'_i replaced a selected meta-neuron q_i, as specified in the philosophy behind DGNN model, we want S'_i to realize a more complex function than the standard one associated to q_i (sigmoid, ReLu, etc.). Going over the concept of activation functions, we want the subnet to realize a function such that the new input-output mapping of the network

will result in a lower output error, or in other words we want S_i' to learn a useful non-linearity that contributes to decrease the loss function values. To do that, we train $S_i^{[k+1]}$ with a dataset $\mathcal{D}_i^{[k+1]}$ where the input patterns $\mathbf{x}^{[k+1]}$ correspond to activations of the replaced depth-k meta-neurons, while the targets $\hat{\mathbf{y}}^{[k+1]}$ are generated with a certain target propagation technique. It means that we want the old input to map on new target values. From this perspective, we want the meta-neuron to grow when the subnet it belongs to has not enough computational power to learn the particular assigned mapping. So basically, given *net*, a partially-trained DGNN having depth k, the operations involved in this step for the generic input and target are the followings:

1. Generation of activations of depth-k meta-neurons, or input to subnets at depth $k+1$:
$$\mathbf{x}^{[k+1]} = F_L^{[k]}(\mathbf{x}) \tag{3.10}$$

2. Generation of targets at depth $k+1$:
$$\hat{\mathbf{y}}^{[k+1]} = \phi_{TP}(\hat{\mathbf{y}}^{[k]}) \tag{3.11}$$
where $\phi_{TP}(.)$ is a target propagation function.

For the particular subnet $S_i^{[k+1]}$, the dataset $\mathcal{D}_i^{[k+1]}$ corresponds to the values of the i-th component of the pair $(\mathbf{x}^{[k+1]}, \hat{\mathbf{y}}^{[k+1]})$. In the next section we discuss the particular target propagation technique implemented by the function $\phi_{TP}(.)$.

The Hybrid Target Propagation Approach

Although any target propagation algorithm might be used, here we rely on RDTP and GB-RDTP (gradient-based RDTP). We know that for construction the RDTP method can be used only when dealing with neural networks having single output unit (and single hidden layer). So when dealing with such kind of architectures, RDTP is the best choice. Anyway, in most cases we handle DGNN with several output units, e.g. a simple classification problem with 3 classes needs a neural network with 3 output units.

So to propagate targets in DGNN with several output unit, we defined the following hybrid technique: when the DGNN has depth $k = 0$, GB-RDTP is used to propagate targets to the hidden layer (to meta-neurons at depth 0), then in the following growing step, the RDTP is used to propagate targets at meta-neurons at depth $k > 0$. We act as mentioned because for depth $k > 1$ we propagate targets to subnets internal layers, and as we know, subnets have single output units, so it perfectly fits for the standard RDTP. In algorithm 6 is shown the pseudo-code for this ad-hoc developed

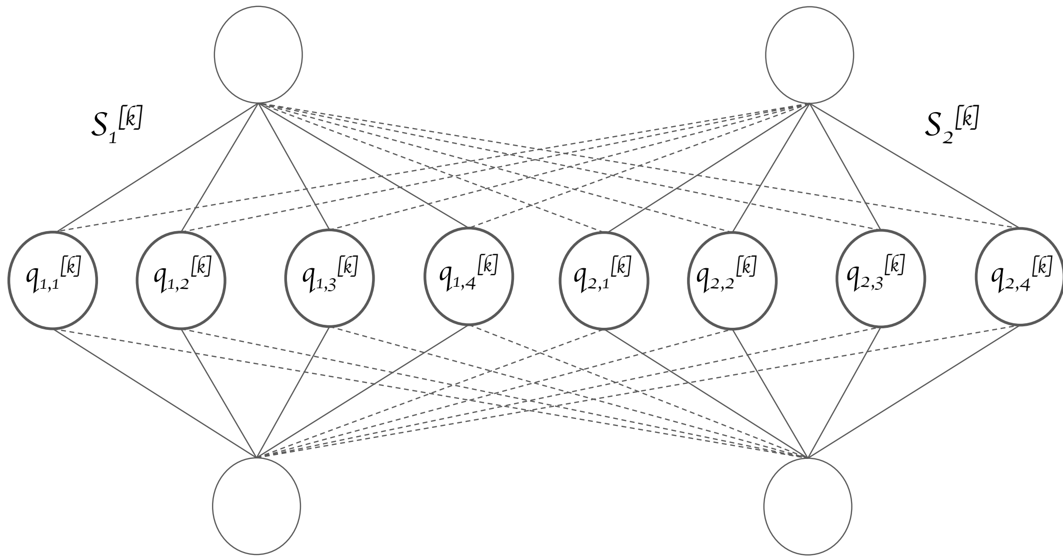

Figure 3.3: Two subnets $S_1^{[k]}, S_2^{[k]}$ are represented as being part of two consecutive standard layer, where the dotted lines represent the connections having fixed weights $w = 0$

hybrid approach, bearing in mind that TP algorithms are meant to propagate targets for all neurons of a certain layer (from layer ℓ to $\ell - 1$), here are intended to propagate targets for all subnets of a given depth (from depth k to $k + 1$).

Algorithm 6 Hybrid Target Propagation

Input: the DGNN *net*, output size *m*, depth *k*, targets $\hat{\mathbf{y}}^{[k]}$.
Output: $\hat{\mathbf{y}}^{[k+1]}$ (propagated targets at depth $k + 1$).

1: **if** $m > 1$ **and** $k == 0$ **then**
2: $\hat{\mathbf{y}}^{[k+1]}$ = `Gradient_Based_RDTP`(net, $\hat{\mathbf{y}}^{[k]}$) ▷ $\hat{\mathbf{y}}^{[k]} = \hat{\mathbf{y}}$
3: **else**
4: $\hat{\mathbf{y}}^{[k+1]}$ = `RDTP`(net, $\hat{\mathbf{y}}^{[k]}$)
5: **end if**

3.4 Parallelizing the Algorithm

In the previous section we have seen which are the steps required to train a DGNN model. Here we analyze those steps in order to find the best way to optimize the training process. As already mentioned, the DGNN framework has a modular nature and this also affects the training algorithm. From previous sections, it clearly emerges to the reader that operations required to grow the single meta-neurons/subnets of the same depth k, most of the time can be performed in parallel. A first optimization step that could be considered is performing the parallel training of involved subnets for a given depth. Going over this approach, with the aim of leveraging massive parallel hardware like graphic cards, here we look at the set of subnets at a given depth k, as if they were nothing but a particular pair of two consecutive standard neural network layers, as indicated in figure 3.3. With this consideration, we can realize the growing step for a set of depth-k subnets by building and training two standard neural network layers, using some ad-hoc strategy. Using this approach, pretending to train multiple depth-k subnets at the same time by training two standard layers, does not allow us to use specific hyper-parameters for each subnet, so we are forced to use the same configuration for all subnets of the same depth-k. So here we can act in a *depth-wise* fashion.

Following the aforementioned considerations, routines used in algorithm 7 and 8 are intended to be acted in a depthwise/layerwise way. Here we give a brief description of them. `Choose_subnet_architectures`(k) returns the number of metaneurons for the subnets at depth k, `Create_new_datasets`(net, k, $\mathcal{D}^{[k]}$) generates the subnet specific dataset for subnets at depth k (all at once),
`Select_metaneurons_to_grow`(net, k, $\mathcal{D}^{[k+1]}$) chooses the metaneurons to replace, `Replace_metaneurons`($\mathcal{P}, \mathcal{A}$) replace the metaneurons with a set of subnets that we call $SubnetLayer^{[k+1]}$ that consists in a pair of standard layer as in fig. 3.3,
`backpropagation`($SubnetLayer^{[k+1]}$) train the subnets (the pair of layer) all at once, using the standard backpropagation algorithm.

Algorithm 7 Growing Step

Input: the depth-k DGNN net, k, $\mathcal{D}^{[k]}$.
Output: the DGNN having depth $k+1$, $\mathcal{D}^{[k+1]}$.

1: $\mathcal{A}$ = `Choose_subnet_architectures`($k+1$)
2: $\mathcal{D}^{[k+1]}$ = `Create_new_datasets`(net, k, $\mathcal{D}^{[k]}$)
3: $\mathcal{P}$ = `Select_metaneurons_to_grow` (net, k, $\mathcal{D}^{[k+1]}$)
4: $SubnetLayer^{[k+1]}$ = `Replace_metaneurons`($\mathcal{P}, \mathcal{A}$)
5: `backpropagation`($SubnetLayer^{[k+1]}$)

Algorithm 8 DGNN Training

Input: The DGNN model $dgnn$, dataset $\mathcal{D}$
Output: the trained/grown DGNN

1: $k = 0$
2: `backpropagation`($dgnn$)
3: **while** stopping conditions != True **do**
4: $dgnn$, $\mathcal{D}^{[k+1]}$ = `Growing_Step`($dgnn$, k, $\mathcal{D}^{[k]}$)
5: $k = k + 1$
6: **end while**

From the previous section describing the training process and from the new concept of "subnets-layer" afore introduced, few considerations on the hyperparameters and good practice to manage them need to be done. The training algorithm requires the selection of new introduced hyperparameters:

- $\gamma \in [0, 1]$ the percentage of meta-neurons to grow.
- ρ the size of the hidden layer of the subnets. It is typically $\sim [4, 7]$ for all depth $k \geq 2$.

Furthermore, as already said before, the configuration of hyperparameters, under this depth-wise approach, is the same for all subnets of a same depth k.

3.5 Experimental Results

In this section we analyse the results obtained by experimenting the techniques/algorithms presented in this chapter. The section contains 3 main experimental setups. In the first one the model is tested on the Vertebral dataset of the UCI repository [16], in the second one on MNIST, the hand written digits images dataset, and in the third one on 3 sub-tasks of the CIFAR-10 dataset, containing small generic images. The first two experiments on the Vertebral dataset are intended to graphically show the model behaviour, while the last one, a 10-folds cross validation procedure, is aimed at confirming the behaviour observed in first two experiments.
For each experiment, and for each of the growing steps we show the learning and generalization curves and the respective accuracy for training, validation and test set. Furthermore, for *depths* greater than zero we show the learning curves for the depth-specific subnet-layer loss.

UCI Vertebral

In the following setting we show how a DGNN model behaves with a real world dataset. Here we deal with the *vertebral column dataset* of UCI machine learning repository [16]. This dataset has been chosen with the goal of comparing the DGNN model with [10], that as said in the previous section, inspired the DGNN idea. The comparison is done at the end of the section. This is a small size dataset, having 310 patterns and 2 classes, where each pattern $\mathbf{x} \in \mathbb{R}^6$.
The next two experiments have the same configuration apart for the ρ value. These two particular configuration show that the DGNN with bigger subnets ($\rho = 6$) is not affected by phenomena that occurs in the smaller-subnet DGNN ($\rho = 5$), as performance deterioration.

DGNN with $\rho = 5$

Regarding the DGNN architecture, input and output sizes are defined by the input and targets dimensionality of the dataset, while the hidden layer is composed of 8 neurons (metaneurons) with sigmoid activation functions. Subnets have one input and output neuron (for construction) and $\rho = 5$ (hidden neurons/metaneurons). The training of the main model (depth-0) is conducted with stochastic gradient descent with mini-batch size of 8, learning rate of 0.01, with cross-entropy loss and sigmoid activation function on the output neurons. For the growing of internal layers ($depth > 1$) we set $\gamma = 0.6$, indicating that 60% of metaneuron must grow (in this case corresponding to $\simeq 5$ metaneurons), and $max_depth = 3$. Lastly, we used Adam as training algorithm for internal layers [33] with learning rate 0.0001. These hyperparameters are selected using a random-search strategy. The dataset is randomly split in training, validation and test set with percentages of 80%, 10%, 10%. In the following figures we show the learning curves for each specific depth. We indicate with "Loss" the loss of the DGNN model, and with "Subnets Loss" the loss of the depth-specific subnet layer, bearing in mind that a set of subnets at the same depth can be seen as a neural network itself.

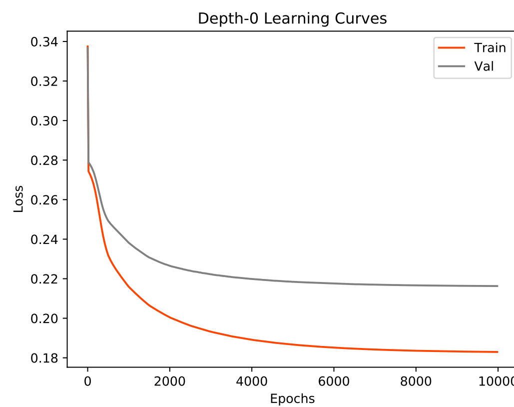

Figure 3.4: Vertebral, depth-0

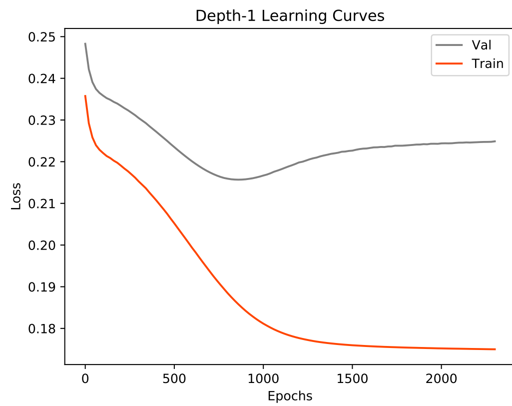

Figure 3.5: Vertebral, depth-1

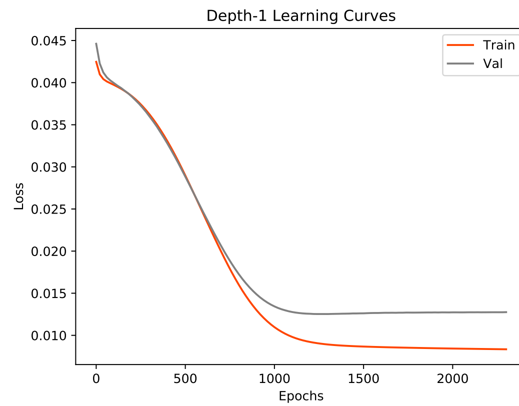

Figure 3.6: Vertebral, depth-1 subnets

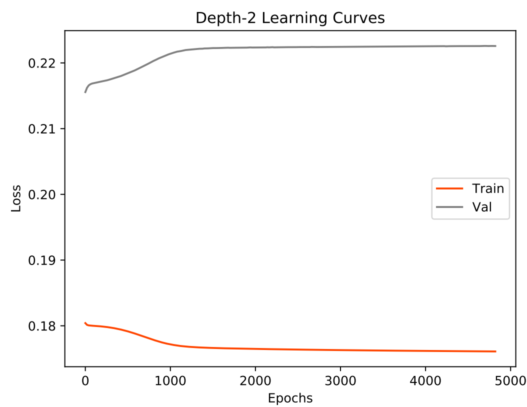

Figure 3.7: Vertebral, depth-2

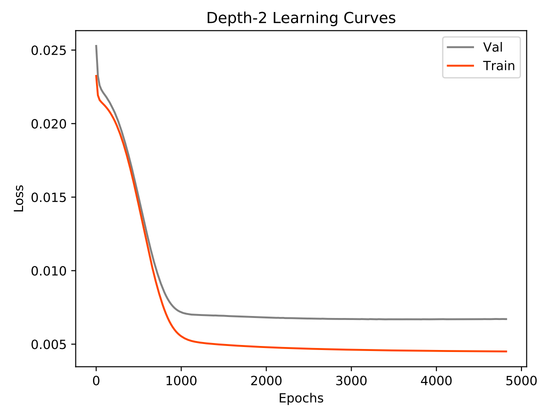

Figure 3.8: Vertebral, depth-2 subnets

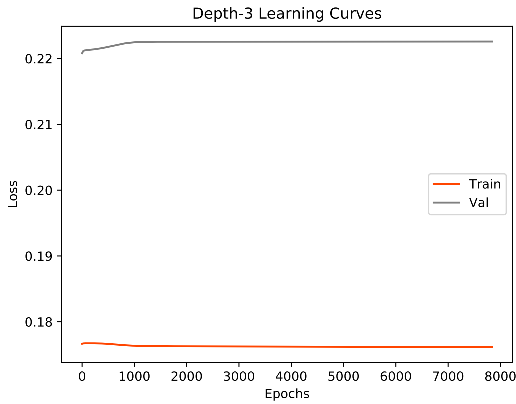

Figure 3.9: Vertebral, depth-3

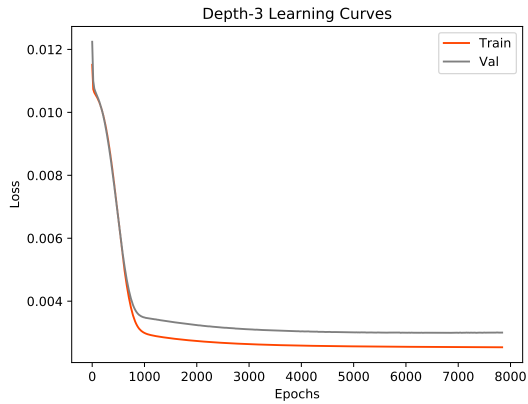

Figure 3.10: Vertebral, depth-3 subnet

We can see that the DGNN loss remains quite the same for depth $k > 1$. The internal loss (the one related to the depth-specific subnet layer) decreases, meaning that the subnets correctly learn from the depth-specific dataset $\mathcal{D}^{[k]}$. Next figures represent the accuracy statistics of the model. Each figure shows the evolution of performances during the training at a specific depth k, ($k = 0, ..., 3$).

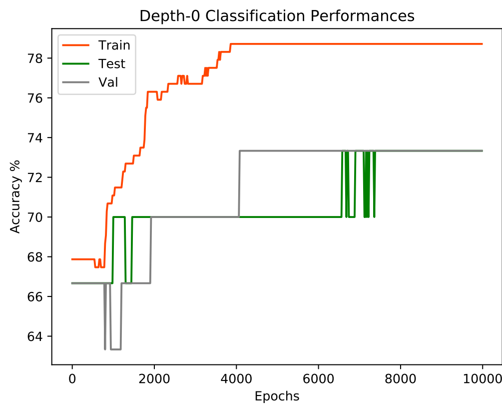

Figure 3.11: Vertebral, depth-0 acc.

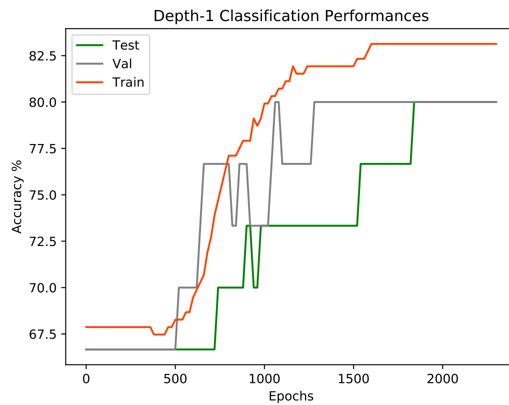

Figure 3.12: Vertebral, depth-1 acc.

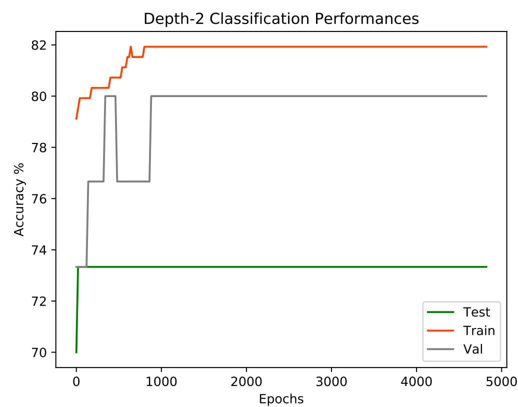

Figure 3.13: Vertebral, depth-2 acc.

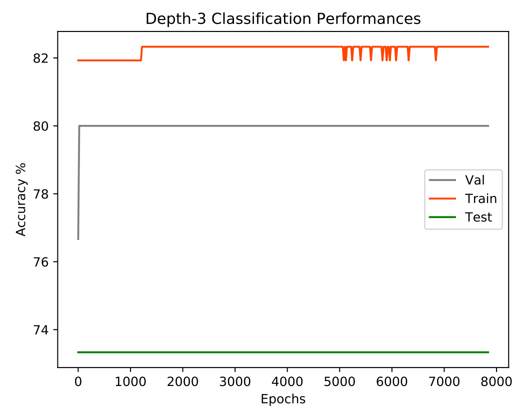

Figure 3.14: Vertebral, depth-3 acc.

From the figures we can observe that there is an improvement of the performances for all curves (train, validation, test) when growing the model from depth-0 to depth-1. The step-wise behaviour for validation and test set accuracy is mainly due to the limited number of patterns belonging to each of them. Training accuracy improves from 78,71% to 83,13%, validation accuracy from 73,3% to 80,0% and test accuracy from 73,3% to 80,0%. In further growing steps, performances do not improve, validation and training accuracies remain quite the same, while we observe a worsening of the test set.

DGNN with $\rho = 6$

Here we use the same configuration of the previous experiment, with the only difference of using subnets with 6 metaneurons ($\rho = 6$) instead of 5. We can see that the phenomena of deterioration seen in previous experiment does not occur, and even if not improving for depth $k > 1$, the statistics remain the same in subsequent growing steps.

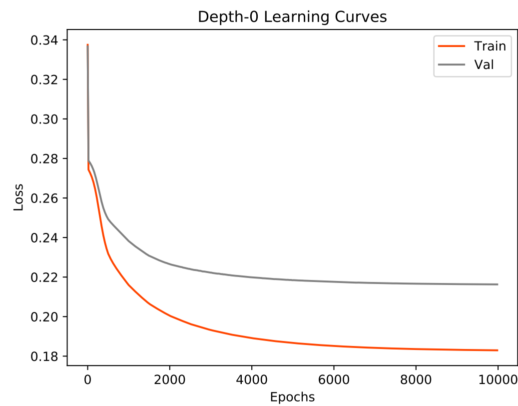

Figure 3.15: Vertebral, depth-0

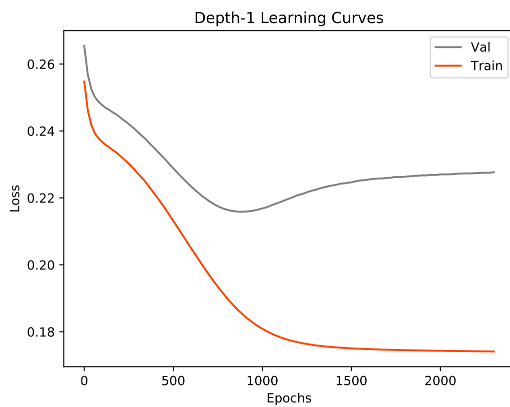

Figure 3.16: Vertebral, depth-1

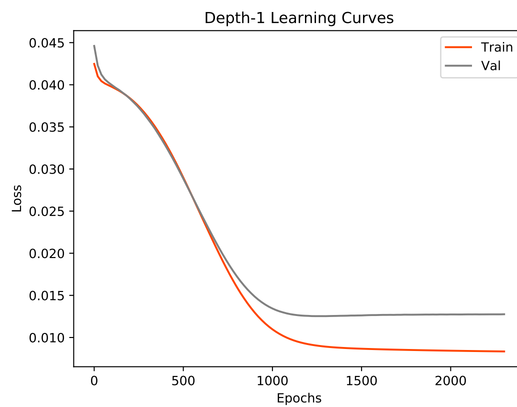

Figure 3.17: Vertebral, depth-1 subnets

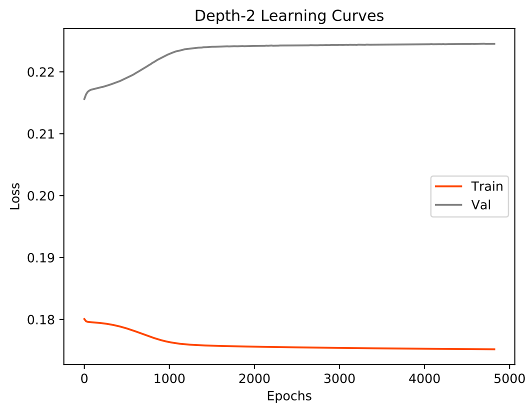

Figure 3.18: Vertebral, depth-2

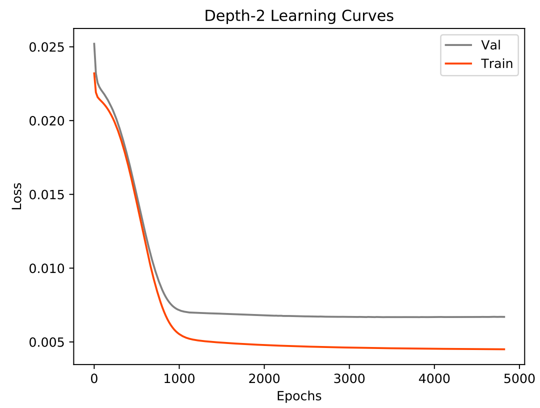

Figure 3.19: Vertebral, depth-2 subnets

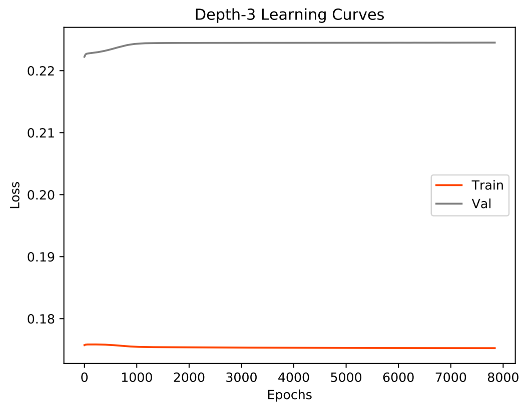

Figure 3.20: Vertebral, depth-3

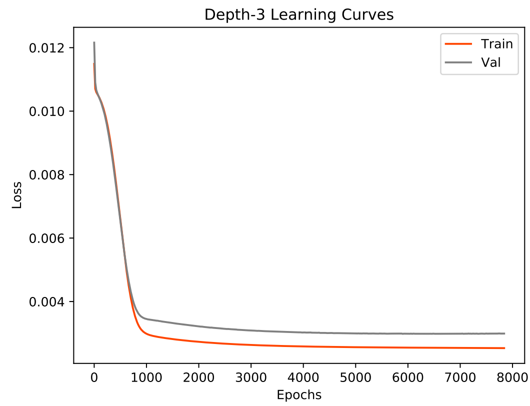

Figure 3.21: Vertebral, depth-3 subnets

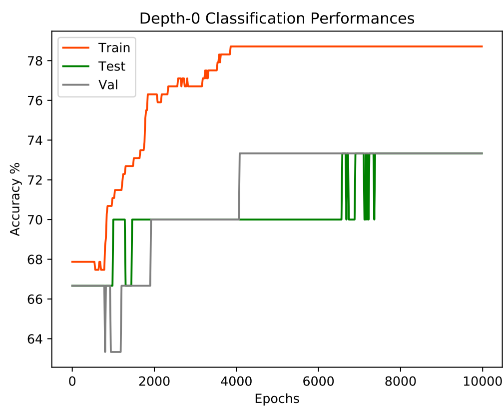

Figure 3.22: Vertebral, depth-0 acc.

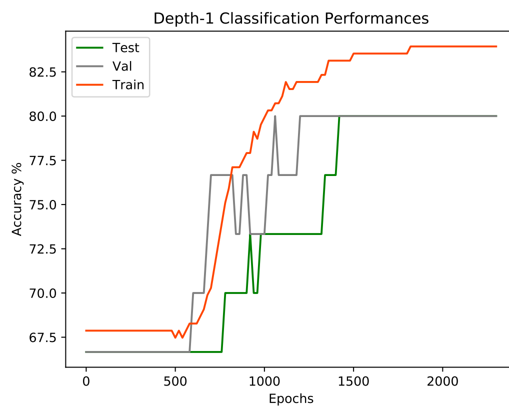

Figure 3.23: Vertebral, depth-1 acc.

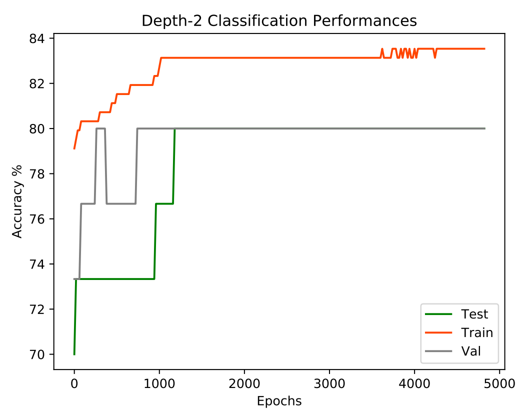

Figure 3.24: Vertebral, depth-2 acc.

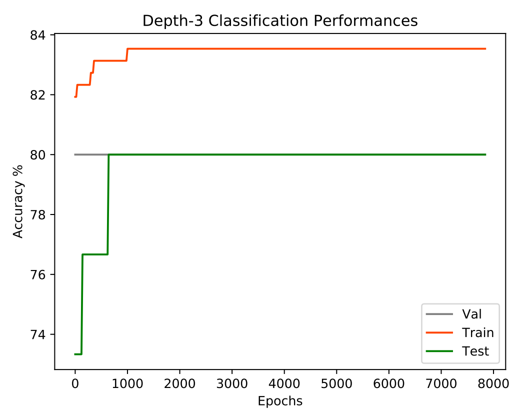

Figure 3.25: Vertebral, depth-3 acc.

10-Fold Crossvalidation

Here we show the statistics of a 10-fold cross-validation experiment, using the same hyperparameter configuration of the second experiment. We plot the accuracy performance for a generic randomly selected fold (in this case the 10-th). An improving can be observed for the train (77,5% to 82,5%) and test performances (82,05% to 84,62%) in the first growing step, while the validation remains constant (77,3%). For the subsequent steps, the performances do not improve.

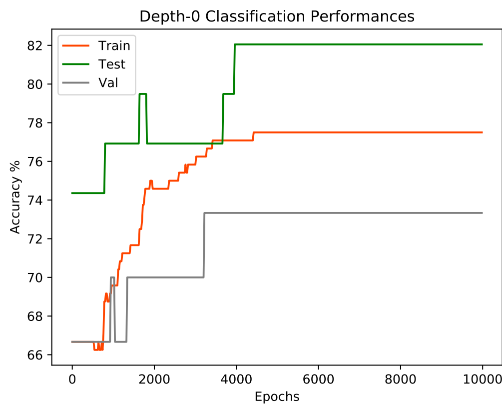

Figure 3.26: Vertebral 10-th fold, depth-0 acc.

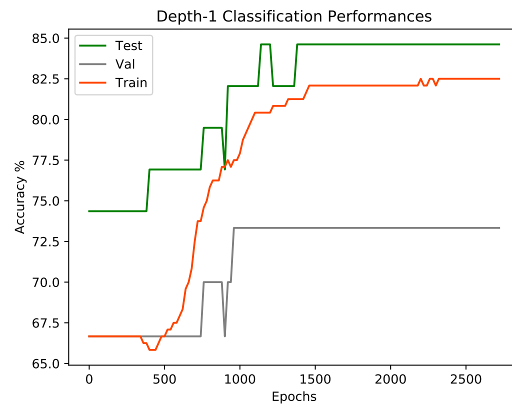

Figure 3.27: Vertebral 10-th fold, depth-1 acc.

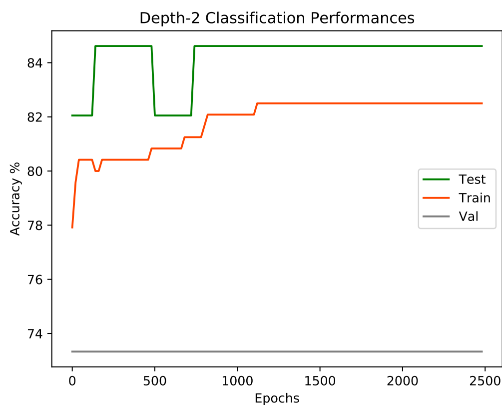

Figure 3.28: Vertebral 10-th fold, depth-2 acc.

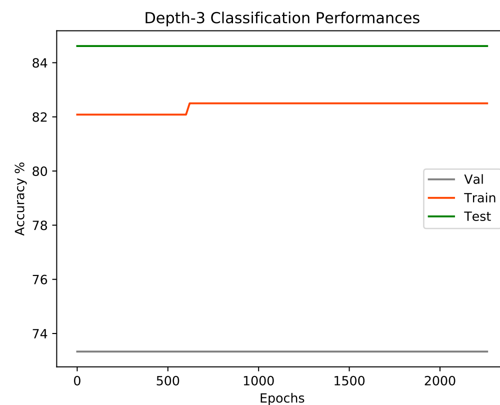

Figure 3.29: Vertebral 10-th fold, depth-3 acc.

In table 3.1 we can see the cross-validation statistics. Although the standard deviation values are very high, the average values indicate that a performance improvement exists. At the end, the performance reached in the last experiment are

Table 3.1: Vertebral 10-fold cross-validation acc. ± std

	Train (%)	*Validation* (%)	*Test* (%)
Depth-0	76,26 ± 3,11	76,28 ± 4,78	75,21 ± 2,56
Depth-1	78.69 ± 3.95	78.79 ± 6.29	77.79 ± 6.10
Depth-2	79.26 ± 4.06	78.54 ± 6.10	77.79 ± 6.10
Depth-3	79.26 ± 4.06	79.21 ± 6.90	77.79 ± 6.10

not comparable with the work of [10]. Anyway we could potentially compare the two techniques from the perspective of the improvement achieved. In [10] the improvement obtained on the test-set corresponds to $0,65\%$ of accuracy, while in this experimental setting it seems to be $2,58\%$.

MNIST

In this section are described results of a DGNN model trained on MNIST dataset. The DGNN is composed of 60 metaneurons, and 10 output neurons with sigmoid activation function. The MSE criterion is used for training the model. Each subnet is composed of 10 metaneurons and only the 60% of metaneurons are grown. The training stops at the second growing step, because not further improving was detected. The training was performed using Adam optimization [33] with a learning rate of 0.001 for all the depths. Due to the size of the dataset, mini-batches of size 32 are used. The improvement of the loss function on validation data was used as stopping criteria, in particular the process ends after 200 steps.

In table 3.2 is shown the accuracy for each single depth. In table 3.3 are shown the loss function values for training and validation data. While there are not improvements on the accuracy performances, we can recognize an improvement on the loss values during the first growing step (from depth-0 to depth-1). In particular, there is an 8.99% improvement on the training loss and a $7,56\%$ improvement on the validation loss. In fig. 3.30 to 3.34 are shown the learning and generalization curves for the each growing step, while fig. 3.35 to 3.37 contain the accuracy values during the training time for the specific depths, for one of the folds.

Table 3.2: MNIST 5-folds crossvalidation accuracy.

	Train (%)	*Validation* (%)	*Test* (%)
Depth-0	98.56 ± 0.16	97.42 ± 0.16	97.13 ± 0.14
Depth-1	98.56 ± 0.14	97.40 ± 0.18	97.12 ± 0.15
Depth-2	98.55 ± 0.14	97.42 ± 0.18	97.11 ± 0.14

Table 3.3: MNIST 5-folds crossvalidation MSE loss.

	Training Loss	*Validation Loss*
Depth-0	$(3.78 \pm 0.2) \times 10^{-3}$	$(5.29 \pm 0.16) \times 10^{-3}$
Depth-1	$(3.44 \pm 0.2) \times 10^{-3}$	$(4.89 \pm 0.15) \times 10^{-3}$
Depth-2	$(3.45 \pm 0.2) \times 10^{-3}$	$(4.90 \pm 0.15) \times 10^{-3}$

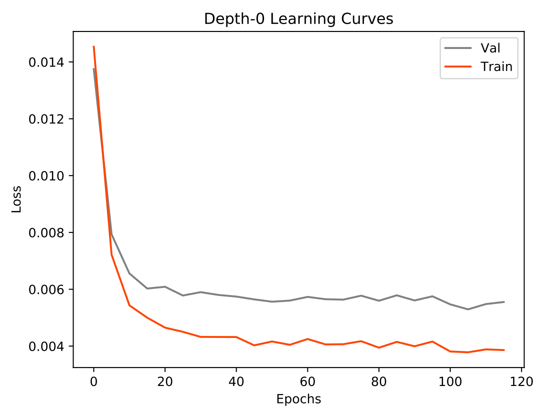

Figure 3.30: MNIST depth-0 loss

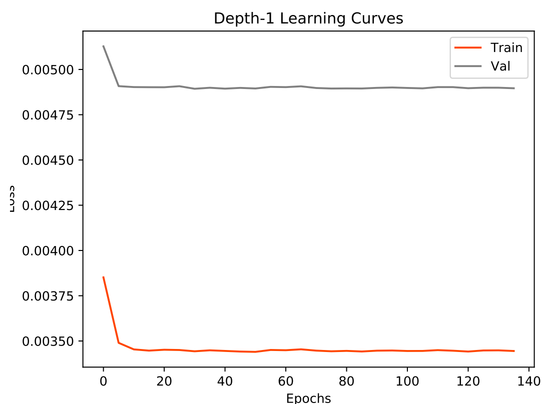

Figure 3.31: MNIST depth-1 loss

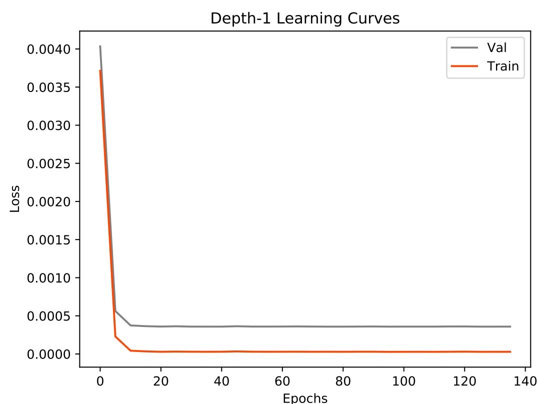

Figure 3.32: MNIST depth-1 subnet's loss

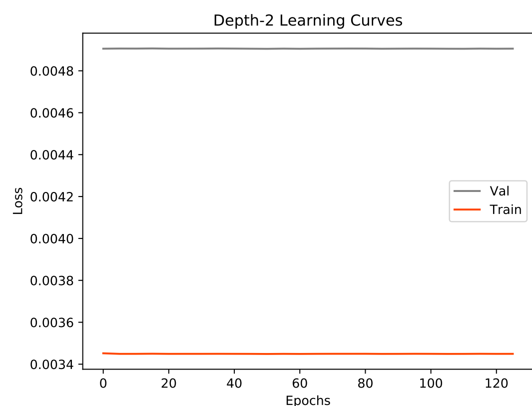

Figure 3.33: MNIST depth-2 loss

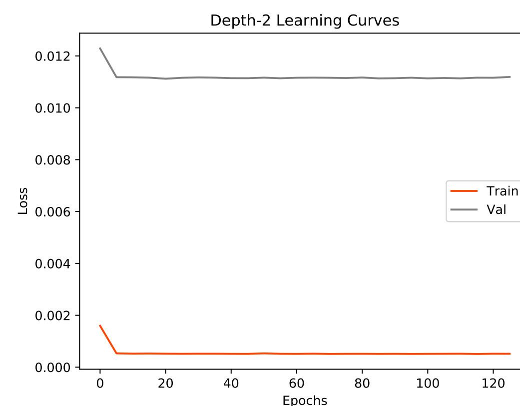

Figure 3.34: MNIST depth-2 subnet's loss

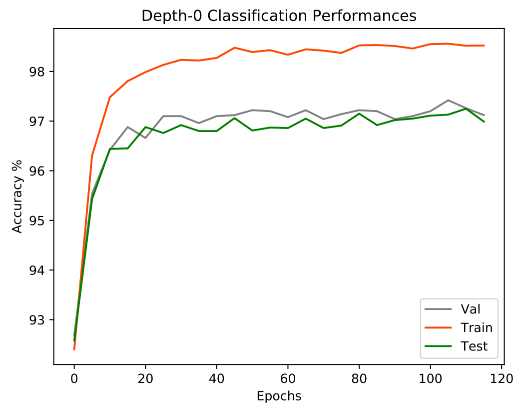

Figure 3.35: MNIST depth-0 acc.

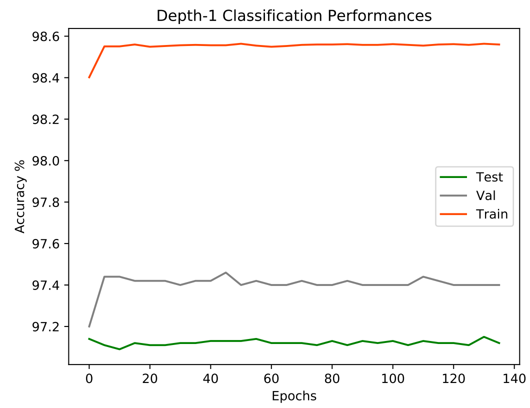

Figure 3.36: MNIST depth-1 acc.

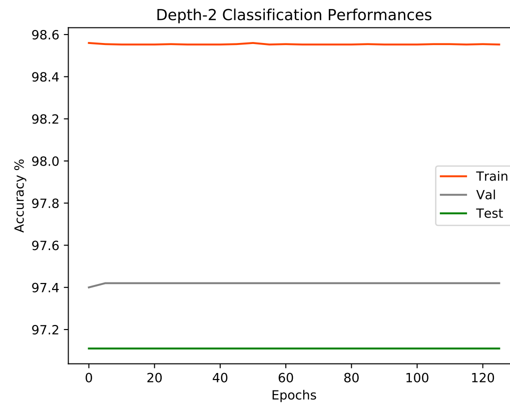

Figure 3.37: MNIST depth-2 acc.

CIFAR-10

In this section we show the performances of a DGNN model trained using subtasks of the CIFAR-10 dataset [35]. CIFAR-10 consists of 60000 images (50000 training - 10000 test) of 32x32 pixels each, organized in 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Each pixel is an integer values in (0, 255) and each image is a tensor of size (3 x 32 x 32), where 3 is due to the RGB channels. It is a well known dataset, mostly used for object detection tasks, but due to its intrinsic difficulty it is also used to test the soundness of machine learning models in general. As the goal of this experiment is to test the DGNN model, and not to reach the state of the art on computer vision tasks, we select patterns belonging to specific pairs of CIFAR-10 classes, reducing in this way the problems to binary classification tasks. In order to face the problem using non-convolutional neural network models, for each image channel, we define a frequency histogram of 40 bins (intervals), and then the concatenation of the 3 histograms, standardized, is used as input pattern. The several classes in the dataset are equally distributed, so each single binary sub-dataset is composed of 10000 training patterns and 2000 test patterns; the 10% of training patterns are used as validation set. For each single sub-problem, a 5-fold crossvalidation procedure is performed.
Knowing that data of the different sub-problems belong to the same dataset, the same architecture has been used for all of them. The DGNN has 50 metaneurons (hidden layer) and a single output neuron, with subnets of size 12, and 60% of meta-neurons growing. Mini batches of size 32 have been used, to deal with the size of the dataset. The chosen sub-problems are the following binary classification tasks: "Deer-Truck", "Deer-Horse" and "Car-Dog". For the first two, a learning rate of 0.001 for all depths has been used, while for the last one, a learning rate of 0.0001 has been chosen for depth-0. Furthermore, for last sub-problem, a weight decay of 10^{-4} is used. Adam optimization algorithm [32] is used for all tasks with an early stopping criterion that stops the training if no improvement is seen on the validation loss after 150 weight-update steps.

"Deer-Truck" Classification

Here we report the performances of the "deer-truck" classification task. The DGNN training ends at depth = 2 because was not seen any further improvement . In table 3.4 the accuracy for each single depth is shown, we can see that on the validation set an improvement exists, even if it is not significative, due to its large standard deviation. In table 3.5 we can observe a significative loss function values reduction, from depth-0 to depth-1, of 7.5% on the training set, and of 7.99% on the validation set.

Table 3.4: Deer-Truck 5-folds crossvalidation acc. ± std. dev.

	Train (%)	*Validation* (%)	*Test* (%)
Depth-0	77.22 ± 0.44	75.37 ± 1.32	73.50 ± 0.64
Depth-1	76.91 ± 0.51	75.53 ± 1.38	73.33 ± 0.56
Depth-2	76.92 ± 0.54	75.57 ± 1.37	73.35 ± 0.51

Table 3.5: Deer-Truck 5-folds crossvalidation MSE loss ± std. dev.

	Training Loss	*Validation Loss*
Depth-0	0.1621 ± 0.0024	0.1758 ± 0.0081
Depth-1	0.1500 ± 0.0028	0.1628 ± 0.0081
Depth-2	0.1500 ± 0.0028	0.1628 ± 0.0081

In fig. 3.38 to 3.42 are shown the learning and generalization curves for the different depths and the relative subnets.

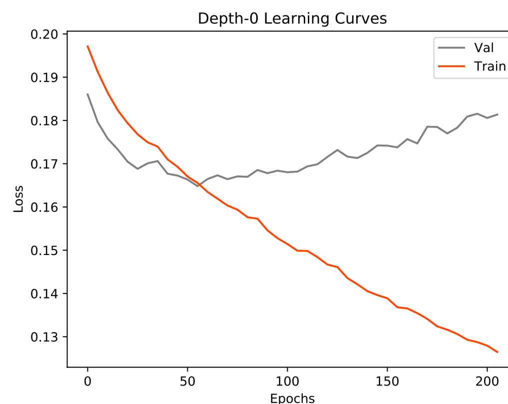

Figure 3.38: Deer-Truck depth-0 loss

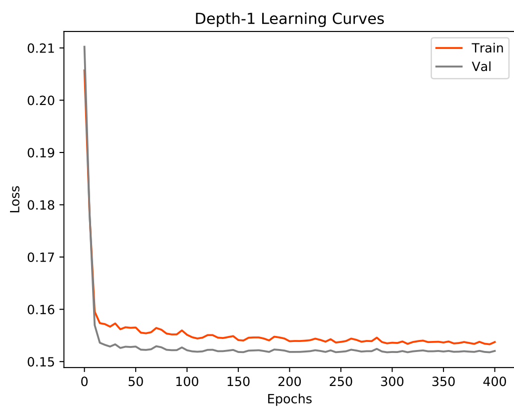

Figure 3.39: Deer-Truck depth-1 loss

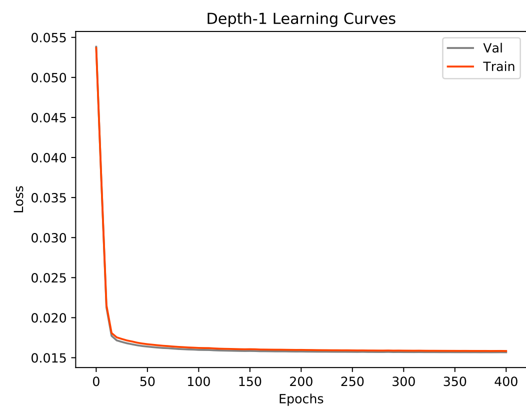

Figure 3.40: Deer-Truck depth-1 subnet's loss

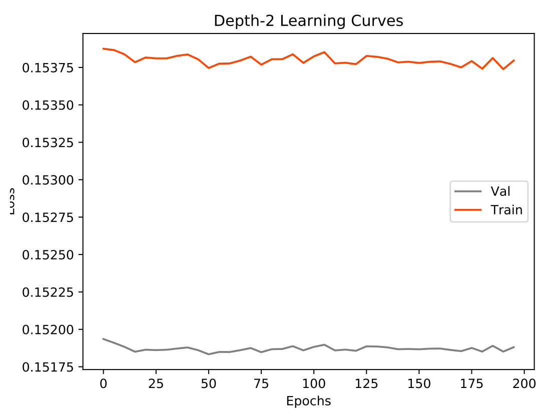

Figure 3.41: Deer-Truck depth-2 loss

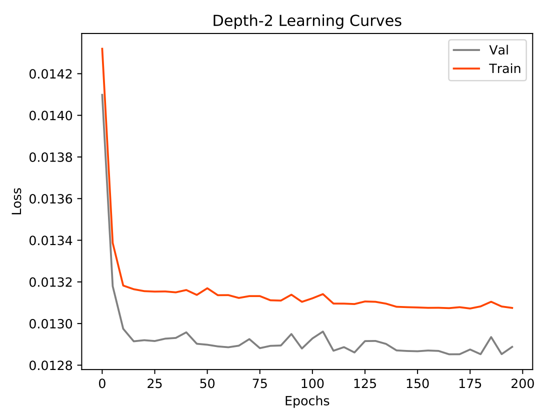

Figure 3.42: Deer-Truck depth-2 subnet's loss

In fig. 3.43 to 3.45 is shown the accuracy for each specific depth.

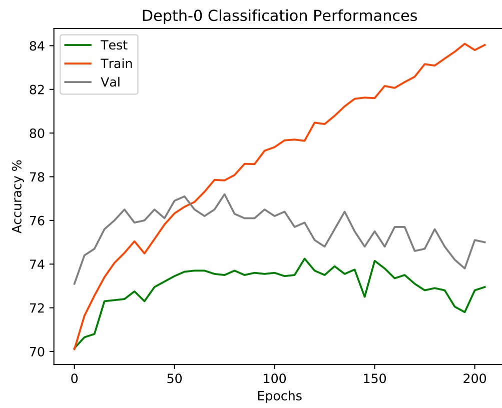

Figure 3.43: Deer-Truck depth-0 acc.

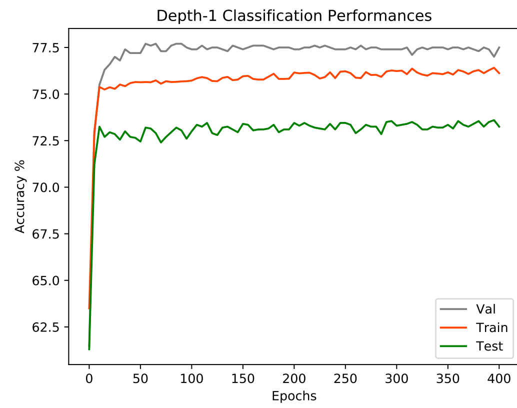

Figure 3.44: Deer-Truck depth-1 acc.

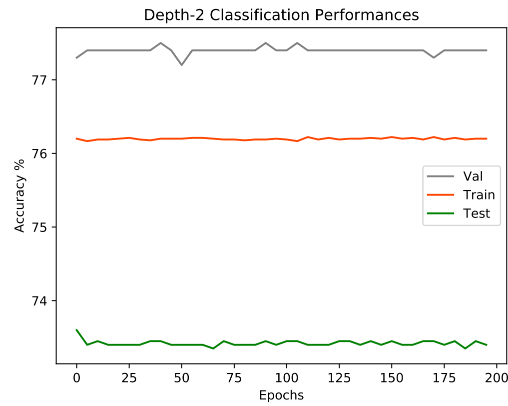

Figure 3.45: Deer-Truck depth-2 acc.

"Deer-Horse" Classification

In this section we show the statistics of the "deer-horse" classification task. From table 3.6, we can see that accuracy does not improve during the model growth, while from table 3.7 we can observe that the MSE loss function values improves (reduces) during the first growing step. In particular we have a $7,6\%$ loss improvement on the training set and of $7,7\%$ on the validation set.

Table 3.6: Deer-Horse 5-folds crossvalidation acc. ± std. dev.

	Train (%)	*Validation* (%)	*Test* (%)
Depth-0	65.58 ± 1.02	61.80 ± 0.41	63.30 ± 0.22
Depth-1	65.31 ± 1.20	61.73 ± 0.59	63.67 ± 0.41
Depth-2	65.39 ± 1.18	61.67 ± 0.34	63.58 ± 0.37

Table 3.7: Deer-Horse 5-folds crossvalidation MSE loss ± std. dev.

	Training Loss	*Validation Loss*
Depth-0	0.2171 ± 0.0035	0.2285 ± 0.0023
Depth-1	0.2006 ± 0.0035	0.2109 ± 0.0023
Depth-2	0.2006 ± 0.0035	0.2109 ± 0.0023

In fig. 3.46 to 3.50 are shown the learning and generalization curves for each single depth and the relative subnet-layers.

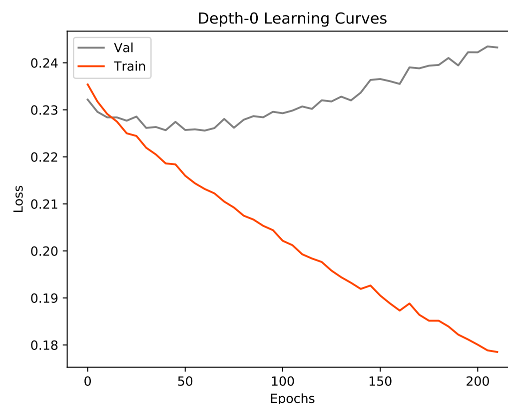

Figure 3.46: Deer-Horse depth-0 loss

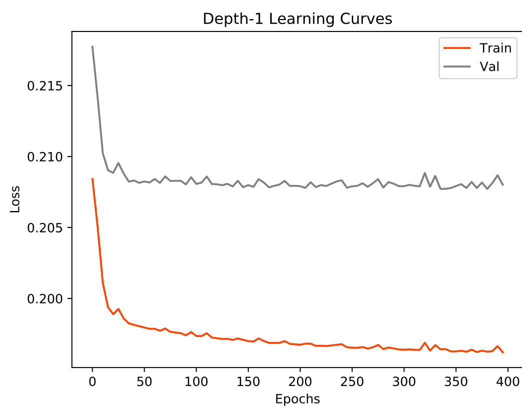

Figure 3.47: Deer-Horse depth-1 loss

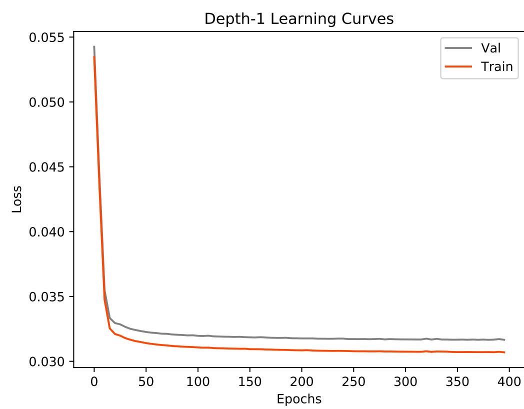

Figure 3.48: Deer-Horse depth-1 subnet's loss

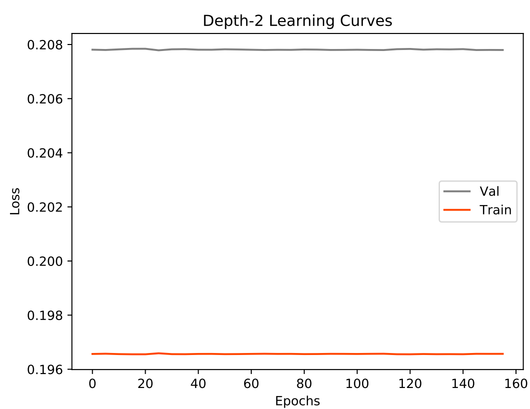

Figure 3.49: Deer-Horse depth-2 loss

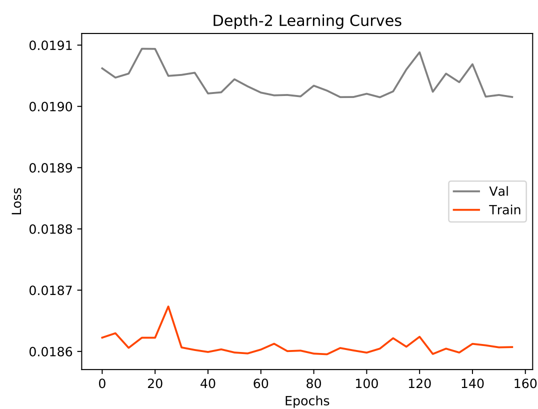

Figure 3.50: Deer-Horse depth-2 subnet's loss

In fig. 3.51 to 3.53 is shown the accuracy for each specific depth.

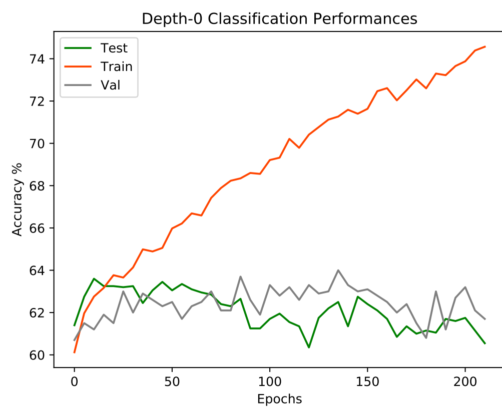

Figure 3.51: Deer-Horse depth-0 acc.

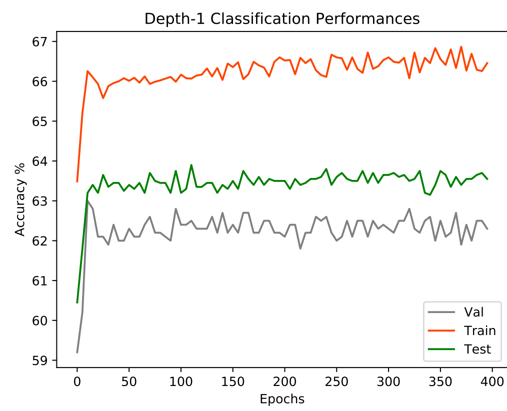

Figure 3.52: Deer-Horse depth-1 acc.

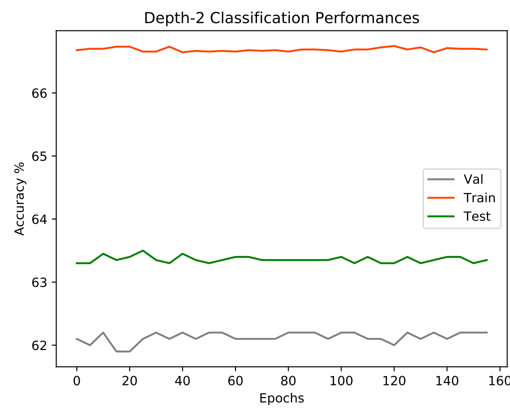

Figure 3.53: Deer-Horse depth-2 acc.

"Car-Dog" Classification

In this section we show the statistics of the "car-dog" classification sub-problem. From the table 3.8 it seems that the accuracy slightly improves over the validation and test data, but it is weakly supported by the standard deviation values. From the table 3.9 instead we can see a loss value reduction on both training and validation set. More in detail, there is a 7.61% improvement on the training loss, and of 7.77% on the validation set.

Table 3.8: Car-Dog 5-folds crossvalidation acc. ± std. dev.

	Train (%)	*Validation* (%)	*Test* (%)
Depth-0	67.20 ± 1.03	62.07 ± 1.19	62.58 ± 0.31
Depth-1	66.84 ± 1.09	62.60 ± 1.36	62.58 ± 0.54
Depth-2	66.90 ± 0.97	62.67 ± 1.22	62.65 ± 0.56

Table 3.9: Car-Dog 5-folds crossvalidation MSE loss ± std. dev.

	Training Loss	*Validation Loss*
Depth-0	0.2103 ± 0.0052	0.2264 ± 0.0034
Depth-1	0.1943 ± 0.0045	0.2088 ± 0.0033
Depth-2	0.1943 ± 0.0045	0.2089 ± 0.0033

In fig. 3.54 to 3.58 are shown the learning and generalization curves for each single depth and the relative subnet-layers.

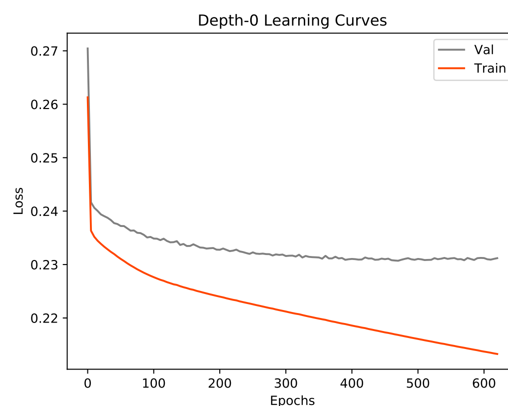

Figure 3.54: Car-Dog depth-0 loss

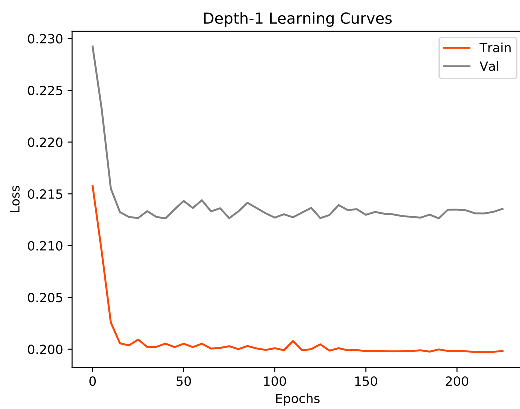

Figure 3.55: Car-Dog depth-1 loss

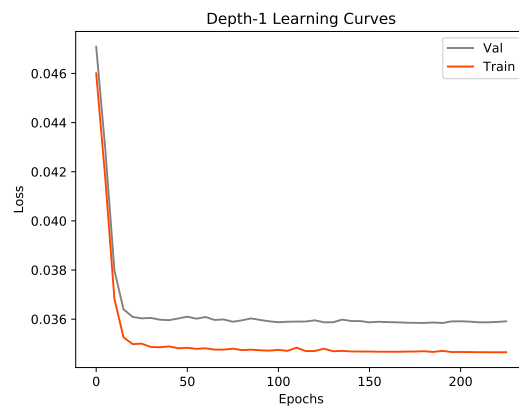

Figure 3.56: Car-Dog depth-1 subnet's loss

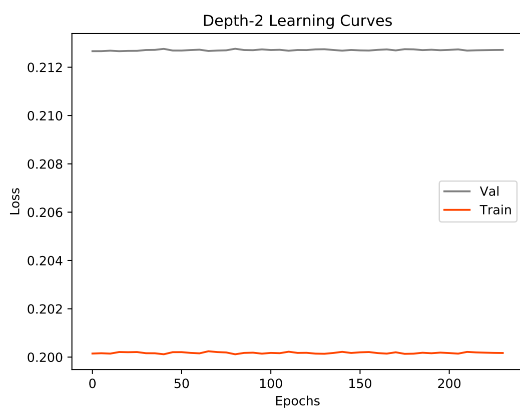

Figure 3.57: Car-Dog depth-2 loss

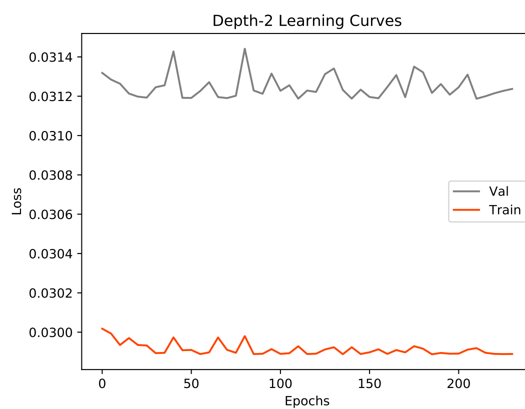

Figure 3.58: Car-Dog depth-2 subnet's loss

In fig. 3.59 to 3.61 is shown the accuracy for each specific depth.

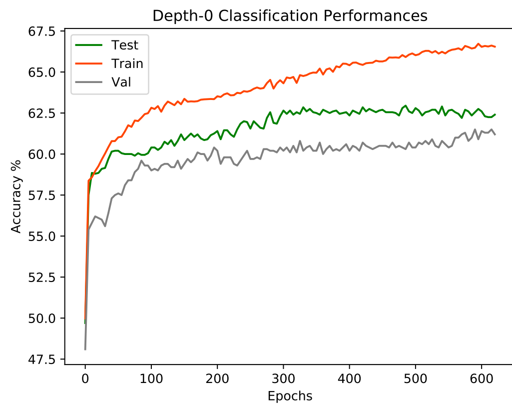

Figure 3.59: Car-Dog depth-0 acc.

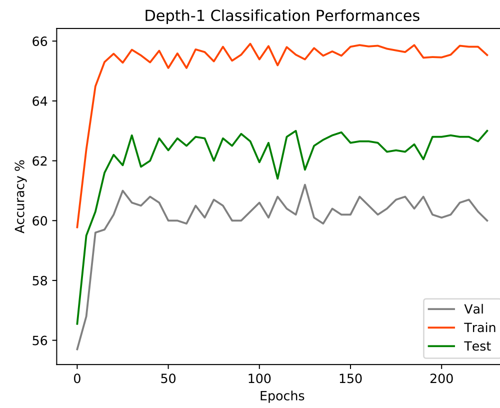

Figure 3.60: Car-Dog depth-1 acc.

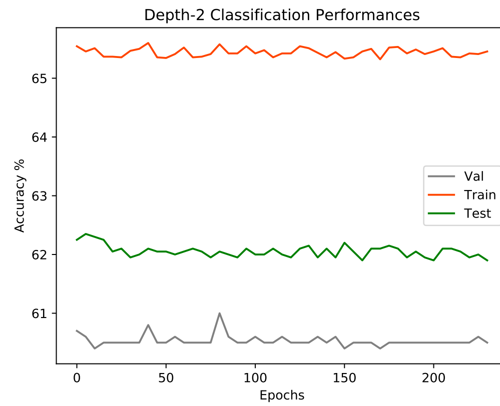

Figure 3.61: Car-Dog depth-2 acc.

Results Considerations

The proposed DGNN model has been tested on the Vertebral UCI dataset, MNIST, and three sub-problems extracted from the CIFAR-10 dataset. It has been seen that the DGNN reaches its goals partially with the Vertebral dataset, but the standard deviations are too high. Anyway in some single experiments, as the one represented in fig. 3.22-3.25, it is clear that the model works, improving the accuracy performances at least on the first growing step. In experiments conducted on much bigger dataset as MNIST and CIFAR-10 the weakness of the framework is even more visible, where no substantial improvement has been seen on the accuracies. On the other way around, analysis of the loss function values clearly demonstrate that a significant improvement exists, even if limited to the first growing step, and it is confirmed by all experiments performed on MNIST and the three sub-tasks on CIFAR-10, and it typically stands around the $7\% - 8\%$ for both training and validation set.
Anyway, it is clear that the loss values can improve (but not too much) without altering the accuracy values. Basically, while the accuracy does not take into account the confidence of a given classification outcome, loss function does. So improving the loss value without altering the accuracy means that the growing step make the model more confident, or in other words make the decision function more fitted to the data points at hand.

3.6 Remarks

In this chapter we introduced the motivations and the idea that led to the development of the DGNN model. We formalized the model, and defined an appropriate learning algorithm. At the end, an extensive experimental section has been conducted.
From experiments it basically turns out that the model works only partially. The growing algorithm is well posed, and in certain scenario it is able to improve model performances. Unfortunately not always. From the other way around, it worths to note that it does not destabilize the learning process of the whole model.
The main limitation seems to be the difficulty of the learning problem generated for the specific internal subnets. Albeit some mathematical tools to assess the difficulty of a learning problem (seen as a function) exists, only empirical investigations have been conducted. Being the input and output layer of the internal subnets composed by a single neurons, we had to opportunity to graphically inspect some of the subnet specific dataset generated. From this first analysis was immediately evident that in some situations, the functions that subnets were required to learn were extremely difficult and would require hundreds of hidden units, without any warranty of convergence.
Being aware of the limitations of the developed model, further research has been conducted to overcome these issues. In the next section, identified solutions are extensively discussed.

Chapter 4

Downward-Growing Neural Architectures

The need of a neural network that autonomously defines its architecture has already been discussed in the chapter 1 and chapter 3 of the present work. From the experimental section of the chapter 3, it appears that the DGNN framework does not reflect the expectations, and few architectural limitations have been highlighted. This context pushed the research activity along new directions, in order to overcome these limitations and continue to pursue the original goal. The solutions identified to solve the DGNN issues then resulted in the definition of a different framework, with different growing strategies, architectures and learning algorithm.
In this chapter we introduce the *Downward-Growing Neural Architecture* (DGNA) framework. Although the main goal remains the same, the perspective from which problem is faced changes. The searching for the right neural architecture becomes the consequence of a different objective, that is the definition of a model whose related separation surfaces become more and more complex, in order to better classify data at hand and improving performances. This is realized by adding further computational units to the architecture, in a well specific manner. This new perspective led us to conceive a different growing strategy.

4.1 Related Works

It worths to say that the idea of let a neural network structure evolve, as in a natural development process is not new, rather it dates back at the same period of the development of the idea of neural network. In this section we discuss few most famous (and related) growing neural network architectures.

Cascade-Correlation. One of the first important work in the area of growing neural architecture is "Cascade-Correlation" neural networks [19]. The idea behind the development of the model is the same that drove our proposed model (DGNA), that is to add further computational units to the model in order to hopefully improve its computational capability and reduce the model errors. At the beginning, a single layer neural network is created, without hidden layers, where the number of input and output units is driven by the nature of the problem, as always. Being the configuration a one-layer model, it does not require the use of backpropagation [49] to define gradients for internal layer, and learning algorithm like perceptron [48] or Widrow-Hoff learning rule [57] can be used. After a first learning phase, if the performances of the model are not as expected, the growing phase starts. It consists in adding a single neuron per time to the model. Once the new neuron is added, each existing neuron in the model (except output neurons) will be connected to the new one, and its input connection weights will be learned by maximizing the correlation between the outcome of the newly introduced neuron and the network output. As a second step, the values of the input connections are frozen and the output weights are learned using a single-layer learning algorithm as before. Then the process is repeated until convergence, bearing in mind that the output of the newly introduced neuron will be served as input to the next added neuron.
This basically generate a deep architecture where each hidden layer is composed of single neurons, and each internal neuron has a direct connection with all previous neurons.

Growing Neural Gas. Another model to consider is "Growing neural gas" [22]. This is an unsupervised learning model, therefore not strictly related to the nature of our proposed model framework. This is an extension of the original "neural gas" model [41], a competitive hebbian-learning [40] based unsupervised network aimed at learning the topology of data. It is basically and incremental version of [41], where during the learning, further computational units are introduced.

Greedy layerwise. A work that deserve our attention, for two reasons, is [4]. The first reason is that it can be considered a growing architecture, second is that it is one of the milestone of the deep learning phenomena. In this work are described

the first techniques that lead the to creation of deep architectures. In particular, algorithms defined in [4] where basically used to pre-train deep neural networks. The method consists in growing a network architecture adding one layer per time, and training the newly introduced layer in an autoencoding like fashion.

Deep Growing Learning. An interesting interconnection between growing models and the semi-supervised techniques is [55]. In a partially labeled setup, as the model learns from the supervised data, it is used to estimates the labels for unlabeled data, augmenting in this way the size of the supervised dataset. In order to face the more and more increasing size of the dataset, it needs more computational power, and further layers are plugged into the network. The new layer is initialized as a copy of the previous one, then a fine tuning procedure is acted.

Dynamically expandable neural network. Within the reasons that drove the research in the area of dynamic architectures (where growing can be considered an instance of this), recently emerged the need of having architectures able to face the problem of continual learning. It means that the learning task may change over time, adding further classes to the problem at hand and/or new data to be classified.
The key idea in [58] is to use the knowledge of an already trained model, to train a further model in order to tackle the new task. Basically there are 3 possible strategies, depending on the difference (in terms of data distributions) between the new task data and the old one. 1. *Selective retraining*: the data belong to similar distributions. One or more output neurons are added. The last layer is trained while keeping fixed all other weights of the model. Then the whole model is retrained, updating only the weights above a certain threshold. 2. *Dynamic Expansion*: the tasks have rather different distributions. The capacity of the network is increased by adding a certain number of further neurons. 3. *Network Duplication*: to prevent the catastrophic forgetting, that is the model begins to worst the performance on the old task. The neurons whose related weights differ more than a value γ (hyperparameter) from the old weights, are duplicated on the same layer. After this duplication process, the network is retrained.

Adanet. Another interesting work to mention is "Adanet" [13]. The idea is always the same, let a given model grows until performances improve. Here the objective to minimize is a trade-off between the model complexity and the empirical risk. The learning procedure is similar to previously seen ones. The model starts in a base configuration, then further computational units are added incrementally. Here they act as follows: given a base model $\mathbf{h}_\ell$ having ℓ layers, they create two candidate network $\mathbf{h}'_\ell$ and $\mathbf{h}'_{\ell+1}$, having respectively ℓ and $\ell + 1$ layers. The added networks can be connected with each unit in $\mathbf{h}_\ell$, leveraging in this way the already existing data

representations. In order to choose the one that better contribute to improve the objective function, both candidate are tested. So the model keeps the learning process first using $\mathbf{h}'_\ell$ and then using $\mathbf{h}'_{\ell+1}$. The one that produce better results is chosen to extend the architecture. As it is easy to image, the computational cost can be very high, especially when dealing with big architectures.

4.2 Growing Architectures as a Search Strategy

With this framework we aim at finding good network architecture configurations using a *growing strategy*. We start with a certain network configuration, that we name *base-model* or *base-network* and then we let it evolve adding other computational units, as emulating a natural growing process. Advantages of these approaches over the well known trial-and-error strategies are that the work done in training the base-model (although if it results in a not very performing model) is not wasted. The growing process leverages the existing architecture (and relative connection weights), using it as a starting point to define more complex models, entailing in this way a very significant time saving.
As described in the related works section, several strategies have been proposed to solve the problem of searching the best fitting architecture, most of them consisted in altering the architecture (adding or removing neurons, adding further layers or even parallel networks as in Adanet [14]) by relying on some specific criteria: improving some generalization objective, getting better reward in a reinforcement learning context, etc. The here proposed solution is to evolve a given architecture by following a different growing paradigm.

Analysis of decision regions

Several growing solutions have been defined over the years, but to the best of our knowledge all motivated their proposed solution leveraging the fact that having a greater number of computational elements can obviously improve the learning capability. Intuitive explanations of how this could happen were only partially seen. In this work we justify, even if only intuitively, the reason that led to the definition of the proposed growth strategy, explaining how and why this is expected to improve the performance of the model.

Given a MLP, what do we expect from a growing process? *"We want the separation surfaces of the individual hidden neurons, that are initially straight lines, to become non-linear, going to better adapt to the data to be separated."*

To answer this question, we deeply analysed the contribution of single hidden neurons in the learning and classification process. To carry on this analysis we begin considering the example case of an MLP having layers L_0, L_1, L_2 where the hidden layer L_1 is of arbitrary size, L_0 (input layer) is of size 2 and L_2 (output layer) of size 1 . The function realized by the MLP is $\mathbf{y} = f_2(f_1(\mathbf{x}))$ where $f_i = \sigma(W_i\mathbf{x} + b_i)$ is the layer-specific function and corresponds to a mapping $R^{d_{i-1}} \rightarrow R^{d_i}$, d_i is the i-th layer size (in terms of units), f_0 is relative to the input layer, therefore it is not considered and $\sigma : R \rightarrow R$ is the elementwise activation function.

Considering a classification problem with two classes $\Omega = \{\omega_1, \omega_2\}$ associated the dataset $T = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^{N}, \mathbf{x}_j \in R^2, \hat{\mathbf{y}}_j \in \{0,1\}\}$ where targets 0,1 indicate classes ω_1 e ω_2 respectively, the goal of the learning process is to let the generic network output $\mathbf{y}$ be as close as possible to the respective target value $\hat{\mathbf{y}}$, given the input pattern $\mathbf{x}$.
We consider all the network neurons as having logistic sigmoid as activation function. Once the training has been completed, we can consider the weights as constants. Omitting the layer-specific notation, the k-th neuron of L_1 realizes the function $o_k = \sigma(x_1 w_{k1} + x_2 w_{k2} + b_k)$, which is basically a logistic regression; that is the k-th component of the related layer-specific function $f_1(.)$.
Depending on the value of the generic data point $\mathbf{x} = (x_1, x_2)$, o_k is valued on the tails or in the middle range of the sigmoid. It is known that when dealing with high-valued weights, for most of data points, o_k is valued on the tails, reaching values very close to 0 and 1. Rethinking this considerations from the input space perspective, that here is $\mathbb{R}^2$, we have a region R_1 where o_k is closer to 1 and a region R_0 where o_k is very close (or closer) to 0. So, what separates R_0 and R_1 are the points where $o_k = \frac{1}{2}$ exactly. More formally, we define the separation surface associated to the k-th neuron of the i-th layer as $\mathcal{S}_i^k = \{\mathbf{x} : F_i^k(\mathbf{x}) = \gamma\}$, where $F_i(\mathbf{x}) = f_i(f_{i-1}...f_0(\mathbf{x}))$, and in case of sigmoid activation function $\gamma = \frac{1}{2}$. Knowing that $\sigma(.)$ is a sigmoid function, this separation region stands when $x_1 w_{k1} + x_2 w_{k2} + b_k = 0$. The latter can be easily rewritten as

$$x_2 = -x_1 \frac{w_{k2}}{w_{k1}} - \frac{b_k}{w_{k1}} \tag{4.1}$$

that is the equation of a straight line having slope $-\frac{w_{k2}}{w_{k1}}$ and offset $-\frac{b_k}{w_{k1}}$.

So, the computation performed by the layer L_1 can be seen as the sets of the outcomes of the neuron-specific logistic regressions, that graphically corresponds at indicating in which of the two neuron-specific decision regions (R_0, R_1) the generic data point belongs to, for each of the single units (see fig. 4.1).

The consideration done for L_1 can be re-proposed for L_2 too. The difference is that L_2 is fed with the outcomes of L_1. Furthermore, being L_2 the output layer, its separation surface corresponds to the separation surface of the whole classifier. In general we can say that a separation surface $\mathcal{S}_i^k$ is function of the separation surfaces defined at the previous layer: $\mathcal{S}_{i-1}^1, \mathcal{S}_{i-1}^2, ..., \mathcal{S}_{i-1}^{d_{i-1}}$. So the output separation surface in this case is $\mathcal{S}_2^k = \phi(\mathcal{S}_1^1, ..., \mathcal{S}_1^{d_1})$, and in simple architectures like the one considered above, from empirical evidence we know that $\mathcal{S}_2^k$ is approximatively a piecewise linear shape, where the linear parts correspond to the separation surfaces generated in the preceding layer. At the end of the day, we have that the separation surface realized by the classifier is approximatively the one composed by the union of pieces of the straight lines identified by the neurons in L_1, as shown in fig. 4.1 left. After

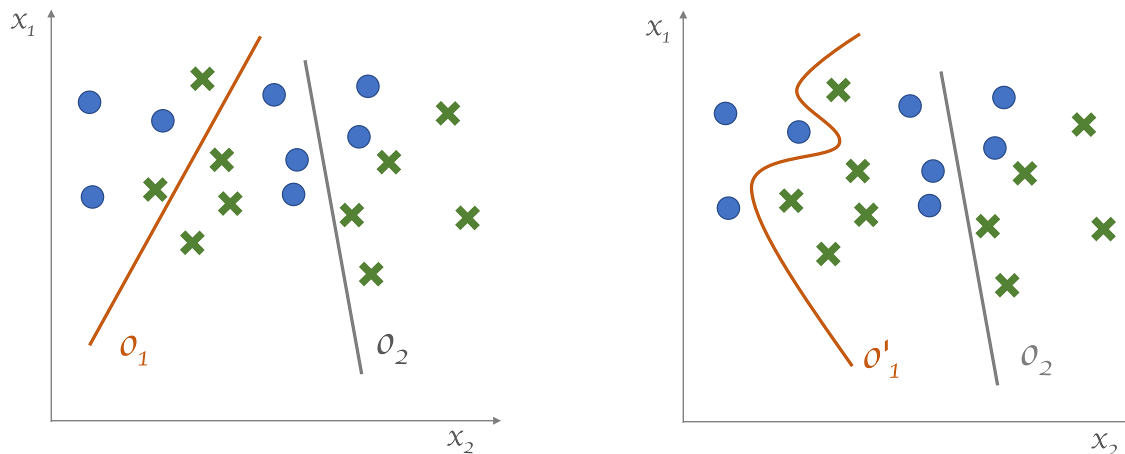

Figure 4.1: This image visually describes what we expect from a growing model. We have a set of points $\in \mathbb{R}^2$ belonging to two classes (cross and circles). We assume a single hidden layer MLP as classifier. **Left**: separation surfaces defined by two hidden neurons. **Right**: separation surfaces expected to be generated by a growing model, where o'_1 is the "evolved version" of o_1.

this discussion the goal is clearer. In most cases, quasi piecewise linear decision regions are not sufficient to separate data efficiently. In this scenario, what we expect from a growing model is the ability to modify its structure/configuration in order to overcome limitations due to the architectural setup. We expect the model to define decision regions adaptable to the model needs, in order to improve performances. The expected behaviour is well described in fig. 4.1.

In case of input spaces having higher dimensionality, let's say m, the outcome of the k-neuron is $o_k = \sigma\Big(\sum_{h=1}^{m} w_{kh} x_h + b_k \Big)$, and the equation realizing the separation surface is $\sum_{h=1}^{m} w_{kh} x_h + b_k = \gamma$, that is an hypercube of dimensionality $m - 1$.

The growing strategy

To increase a network architecture we could define two simple "brute force" techniques: adding neurons to the layer and adding further layers to the network. Most of these techniques have already been tested, and it is clear that adding computational units to the model can be a benefit for the performances. Anyway, some drawbacks exist for the two simple strategies listed above:

1. Even if a single hidden layer FFN (feedforward network) can compute any function [15] with an appropriate number of hidden units, adding units indefinitely to the hidden layer can increase the difficulties for the learning problem.

2. Adding further layers to the model, can theoretically improve the learning capability, but it works only if the first hidden layer has a sufficient number of units [45].

From the analysis carried out in the previous section, it is clear how each unit in the internal layer contributes to the classification process in a simple one hidden layer network. Here we are going to define a growing process strategy, leveraging the considerations that emerged from the previous analysis. Our solution aims at solving at the same time both the two above mentioned issues.

From the graphical perspective described in previous paragraphs, what we expect from a growing model is the evolution of the separation surfaces S_k, with a process that let it become non-linear, so able to define more accurate separation regions.
Knowing that the linear decision surface $\mathcal{S}_i^k$ is the set of points where o_k is equal to a certain value γ (in case of sigmoidal activation function $\gamma = \frac{1}{2}$), and knowing that the function realized by o_k is $o_k = \sigma(\mathbf{w}_k^{in}\mathbf{x} + b_k)$ that depends on the neuron input weights indicated as $\mathbf{w}_k^{in} = (w_{k1}, w_{k2})$ and the bias, with $\sigma(.)$ the generic activation function, the idea is to replace the neuron and all its input connections $(\mathbf{w}_k^{in}, b_k)$ with a more powerful processing component realizing a non-linear function $\varphi : \mathbb{R}^{d_0} \to \mathbb{R}$, such that the related decision surface $\mathcal{S}^k$ will result in a more complex and adaptable shape. This more powerful processing component in turn is a smaller neural network that we call subnet Sub. After this step, we have that $o_k = \varphi(\mathbf{x}; \mathcal{W})$, where $\mathcal{W}$ represent all the weights of the subnet.
This procedure aimed at realizing more suitable decision regions, also entails a modification of the original network architecture, that it is gradually adapting to computational needs, as in a sort of growing process.

The above defined process can be realized for each neuron of L_1. In this way, at the end of the growing process, we have that the network developed a further internal layer. As a consequence, with this techniques we have that the original first-layers completely changed. Albeit the growing strategy has been introduced using a two dimensional input space context as sample, it works for any input dimensionality. Furthermore it is easy to infer that the procedure can be repeated multiple time, defining in this way a network architecture with an arbitrary number of internal layers. This process as a whole, can be seen as a possible solution to the problem of neural architecture search.
In the next section we describe how to let the subnet Sub_k develop a weights configuration that can realize the needed function $\varphi(.)$ useful to our goals.

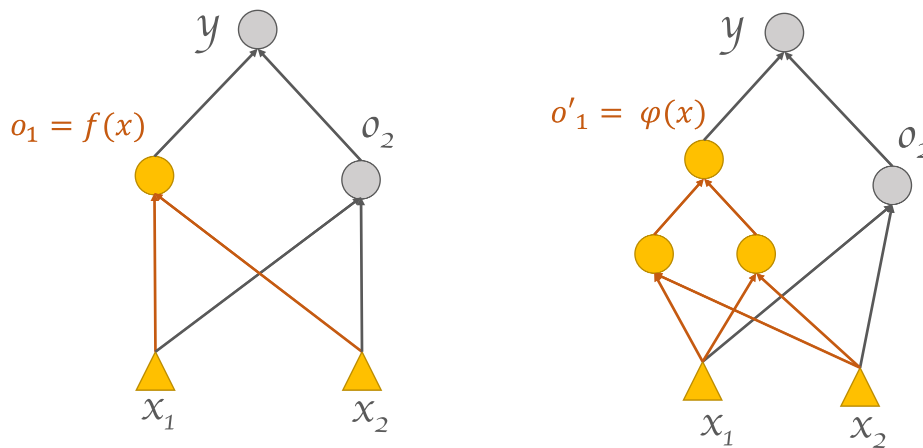

Figure 4.2: **Left**: standard 1 hidden layer feedforward network (FFN), or base-network. **Right**: the grown network, after replacing the leftmost neuron and its input connection with a subnet.

4.3 The Learning Algorithm

In this section we describe the learning algorithm of the proposed growing model. We basically define the downward-growing neural architecture model, named DGNA for convenience, as a neural network having a single hidden layer, with the particularity that its structure can evolve in order to improve the model performances. As already discussed in the section regarding the growing process, we see the gradual evolution of the network architecture as a consequence of the will of defining more complex decision regions (based on the needs). As briefly discussed in previous sections, the growing process can be recursively applied to the model, in order to define neural architectures having an arbitrary number of internal layers, avoiding in this way the well known issues to deal with when training deep neural networks as in the usual way [23, 6].

The search for the right neural architecture is still an unregulated practise, any kind of connection within any units in the network and within any layers (even non-consecutive) is admissible. As already said, we may decide to run the search by adding more units in the hidden layer, since from [15] we know that a one hidden layer network is able to compute any function, with an appropriate number of hidden neurons. Unfortunately, this solution has some limitations: first we may need an

enormous quantity of neurons, and second it is well known that training network with huge hidden layers can be very difficult, or even worse, the training process may not converge.
So we choose to increase the computational capability of the model by adding units in a depth-wise fashion, knowing from [44] that deep network can realize more complex functions than shallow nets requiring smaller layers.

Neural Architecture Search: The Growing Algorithm

Here we describe the DGNA learning process, using the tools ad-hoc realized, described in preceding sections. We can see the neural architecture search as a sequence of growing steps, where each growing step is composed of the following sub-steps, that are the core of the algorithm:

1. **Base-training**: the classical process of training a one hidden layer feedforward network with backpropagation.

2. **Neuron and connections replacement**: replace neurons in L_1, and their specific input connections, with subnets $Sub_k,\ k = 1, ..., d_1$, and training of the subnets.

3. **Refinement**: end-to-end retraining of the grown neural architecture.

The first step to take in order to solve a given learning problem T is the definition of a certain neural architecture (FFN), hence this network is trained with backprop as in the usual way. This is what we call the *base-training* sub-step. At this point it may happen that the performances of the model does not meet our expectations, so we let the model evolve, according to the growing technique described in the previous section: the neurons of the hidden layer and their input connections are replaced with subnets. These subnets are then trained in turn, so as to be able to carry out more complex functions (and related surface decisions). This is the real growth step. As a last sub-step of the algorithm, it is to train the new architecture altogether, so as to align all the weights to optimize the general function (main). This sub-step is said *refinement*.

So we made sure that a simple MLP would grow of one level of depth, in practice by replacing the neuron and its specific input connections with a subnet, for each neuron of the layer. Thus now we find ourselves with a network having two hidden layers. Obviously it can happen that even this architecture does not meet our requirements yet, so we can think that the new architecture can also grow further. To proceed using a recursive approach, in terms of algorithm and implementation, we can think of the subnets generated in the previous step as a single network, i.e. a

single layer FFN, to which we apply the same growth process as in the previous step. Obviously in this case the *base-training* must be considered as already done, and the algorithm begins directly from the second point (neuron and connection replacement). The algorithm 9 represents the process described above; most of the procedure used in the pseudo-code are easily associated with the ones described above, while `check_stopping()` make sure that the performances are improving and that the max allowed depth is not reached, otherwise the algorithm is terminated. The algorithm 10 indicates how to train set of subnets more efficiently, a topic that is discussed more in details in next sections.

Algorithm 9 Neural Architecture Search

Input: $X, Y, max_depth, n_subnets, subnet_size$
Output: the trained Downward-Growing Neural Architecture $dgna$

1: **for** i from 0 to max_depth **do**
2:     **if** $i == 0$ **then**
3:         $dgna$ = create_dgna(in_dim, out_dim, h_dim)
4:         $dgna$ = backpropagation($dgna, X, Y$) ▷ Train the base-network
5:         $Y_T^{(i-1)} = Y$
6:     **else**
7:         $dgna^{(i)} = SubLayer^{(i-1)}$
8:     **end if**
9:     $Y_T^{(i)}$ = estimate_targets($dgna^{(i)}, X, Y_T^{(i-1)}$)
10:     $SubLayer^{(i)}$ = create_subnets($n_subnets, subnet_size$)
11:     $SubLayer^{(i)}$ = train_subnets($SubLayer^{(i)}, X, Y_T^{(i)}$)
12:     $dgna^{(i)}$ = replacement($dgna^{(i)}, SubLayer^{(i)}$)
13:     $dgna^{(i)}, loss$ = refinement($dgna^{(i)}, X, Y_T^{(i-1)}$)
14:     **if** check_stopping($loss, max_depth$) **then**
15:         **break**
16:     **end if**
17: **end for**

Containing the overfitting: subnet-layer training algorithm

Here we describe the procedure defined to train set of subnets in parallel. With the term *subnet-layer* we indicate the subnets that replace more neurons of the same hidden layer. The reason behind the need of defining an ad-hoc training procedure is that of preventing from overfitting issues. It is mainly due to the significant increase in the number of parameters that the model may incur. Training the subnet-layer practically consists in training a set of subnets. Although this can be done sequentially or in parallel (for optimization issues), it consists in solving a certain number of different training problems, each with a its own specific dataset. So to ensure that

the general model is not affected by overfitting, we must ensure that even the subnets are not. A way to reach the goal is to guarantee that each single subnet protects its own generalization capabilities by applying an early stopping criteria independently from all other subnets.
The idea is to define a two-steps training process. The first step consists in training each single *active* subnet for a single epoch. We say that a subnet is *active* if it does not meet the stopping conditions yet. Subnets whose performances improved after this single training epoch are added to the "proposed-subnets" list. The second step consists in evaluating performances of the whole model using the "proposed-subnets" list as *subnet-layer*. If performances of the main model improve, than the "proposed-subnets" becomes the new "subnet-layer". The procedure is repeated until convergence, as described in the algorithm 10.

Algorithm 10 Subnet-Layer Training

Input: X, Y, Y_T, $main_net$, $dgnn$, $SubLayer$
Output: The trained subnet-layer

1: $proposed_subnets = SubLayer$ ▷ Initialization
2: **for** e from 1 **to** max_epoch **do**
3:     **for** k from 1 **to** $n_subnets$ **do** ▷ 1 epoch of training for each subnet.
4:         $Sub = SubLayer[k]$
5:         Sub, $loss_e$ = train_epoch(Sub)
6:         check_stopping_condition(Sub)
7:         **if** $loss_e < loss_{e-1}$ **then**
8:             $proposed_subnets[k] = Sub$
9:         **end if**
10:     **end for**
11:     $loss_proposed$ = evaluate_with_layer($main_net$, $proposed_subnets$)
12:     $loss_actual$ = evaluate_with_layer($main_net$, $SubLayer$)
13:     **if** $loss_proposed < loss_actual$ **then**
14:         $SubLayer = proposed_subnets$
15:     **end if**
16:     check_stopping_condition($main_net$)
17: **end for**
18: return net

Dealing with many neurons: subnets weights sharing

When dealing with hidden layers having many neurons, substituting each neuron and its input weights with a subnet and then train all the subnets can become unfeasible. To overcome this issue, it was thought to share the weights of multiple subnets. In this way, supposing we want to replace k neurons and the relative input connections, instead of creating k subnets, each having single output and a certain

number h of hidden neurons, we may create a single subnet having h hidden neurons (or even more) and k output neurons.
To contain the number of hyperparameters, we define the following *good practice* for the architectural choice of subnets. In order to reduce the search space, first we impose some constraints on the type of architecture that we want to generate. Typically, deep architectures are *rectangular*, where all internal layers have the same size, or *pyramidal*, where each layer has less neuron than the previous one. Assuming to choose the rectangular architecture, we automatically impose the constraint that the sum of the hidden neurons of the replacing subnets must be equal to the size of the hidden layer of the main network. Therefore, the only remaining hyperparameter is the number of subnets.
In this regard we need only to pay attention in choosing the right number of hidden units for the base-network such that it must be divisible by the number of subnets. It is generally recommended from a minimum of 2-3 to a maximum of 10. The choice need to be done also considering the dataset size, where using fewer subnets can results in a significant time saving.

4.4 Experimental results

Here we describe the experiments that validate the effectiveness of the downward-growing neural architecture (DGNA) model. Experiments have been designed to prove firstly the effectiveness of the proposed algorithm, and second to demonstrate that the achieved results are comparable to the state of the art, and sometimes are even better.

UCI Datasets

We have chosen the following datasets, in order to span through different characteristics that can profile a dataset, as number of features, number of output classes, and number of patterns. The datasets involved in the experimentations are: Adult, Ozone, Ionosphere, Wine, Vertebral, Blood; their main characteristics are shown in table 4.1.

Table 4.1: Characteristics of the datasets used to validate our model.

	Adult	Ozone	Ionosph.	Pima	Wine	Vertebral	Blood
# EXAMPLES	48842	2536	351	768	178	310	748
# FEATURES	14	72	34	8	13	6	5
# CLASSES	2	2	2	2	3	2	2

Those are taken from the work of [20], where 121 UCI datasets [17] were reorganized to run a massive experimentation in order to compare performances of most existing machine learning classifiers, thus releasing a ranking based on their performances. Each dataset is first organized in train-test fold pairs to be used to find the best hyperparameter configuration and second in 4 different folds, to be used for a cross-validation purpose. We chose this setup because it has been used in several works as [34, 46] and many others, stating it as a good dataset collection to be used for comparison, especially for general purpose models.
We than compare our results with those obtained in [34], where authors compared their proposed model with several existing neural network models, on all the 121 UCI datasets, generating in this way a broad and varied benchmark.

All our experiments have been conducted in the same way. First, the hyperparameters (obviously except the number of layers) have been optimized on the predefined validation set, then the model has been evaluated using the 4 predefined folds (3 for training, from which we use 10% for validation, and 1 fold as test set). Training is stopped if the validation loss does not improve at least of 2% after 200 epochs.

Hyperparemeters have been chosen using the random-search strategy [7], selecting the configuration having the best validation loss.
For each single dataset we report the loss and accuracy for each step of the growing process.

Table 4.2: DGNA considered hyperparameters

Hyperparameter	Considered values
# HIDDEN UNITS	{8,16,32,40}
# HIDDEN LAYERS	Autonomous
LEARNING RATE	{0.1, 0.01, 0.001}
LAYER FORM	Rectangular
WEIGHT DECAY	{0.01, 0.001, 0.0001}
# SUBNETS	{ 4 to 20, *#hidden units*}
SUBNET HIDDEN UNITS	$\frac{\#hidden\ units}{\#subnets}$

Ionosphere

Description The Ionosphere dataset is composed of 351 examples of 34 features each and 2 classes. It contains radar data indicating the presence or not of free electrons in the ionosphere. In the following tables are reported the accuracy and loss values for training, validation e test set for each of the growing step. Then the total improvement is reported in the bottom.
In tables 4.3 and 4.4 it is evident how results benefit from the growth process. In this particular case it is interesting to note also how the std.dev. values becomes more and more smaller during the growth. Accuracy values on training validation e test set improve of 5.19%, 6.48% and 5.69% respectively, with an error reduction of roughly 46,6% on the test set if compared to the base-training. We see a consistent improvement in the loss values, with a reduction of 95.77%, 56.12% and 48.82% for training, validation and test sets respectively.
At the end, our model reaches 93.47% accuracy on the test-set.

Table 4.3: Accuracy values for the different growing sub-steps and relative average improvement.

	Train (%)	Validation (%)	Test (%)
BASE TR.	94.70 ± 3.82	88.89 ± 5.86	87.78 ± 2.03
REPL.	99.15 ± 0.99	92.59 ± 3.70	92.90 ± 1.24
REFIN.	99.89 ± 0.18	95.37 ± 1.60	93.47 ± 0.49
IMPROV.	5.19%	6.48%	5.69%

Table 4.4: MSE loss values for different growing sub-steps and relative average improvement.

	Train	Validation	Test
BASE TR.	0.0567 ± 0.0453	0.0989 ± 0.0428	0.1079 ± 0.0206
REPL.	0.0095 ± 0.0119	0.0464 ± 0.0161	0.0602 ± 0.0087
REFIN.	0.0024 ± 0.0033	0.0434 ± 0.0149	0.0563 ± 0.0047
IMPROV.	95.77%	56.12%	48.82%

Architecture and hyperparams: The base network is composed of 32 hidden sigmoid units with one output unit; the growing step has been realized using 4 subnets, each having 20 internal hidden units and 8 output units. Three learning-rates have been used, one per sub-step: 0.01, 0.1, 0.01, and relative L2 penalty: 0.01, 0.001, 0.0001.

Vertebral

Dataset: The vertebral column dataset in its binary classification version, is composed of 310 patterns and 6 features each. It contains biomechanical data related to orthopaedic patients, classifying them as normal or abnormal.
Result: In tables 4.5 and 4.6 are shown the accuracy and loss values for training, validation and test set for each of the growing step. The total improvement is reported at the bottom of each table. It is clear how performances improve both from the accuracy and loss perspective for each of the growing step. The model reaches an accuracy of 87.01%, with an improvement of 8.11% with respect to the base model.

Table 4.5: Accuracy values for different growing sub-steps and relative average improvement.

	Train (%)	Validation (%)	Test (%)
BASE TR.	81.46 ± 1.19	80.21 ± 6.83	78.90 ± 2.96
REPL.	87.44 ± 2.23	82.29 ± 8.53	87.01 ± 2.05
REFIN.	88.64 ± 2.56	84.38 ± 6.83	87.01 ± 1.84
IMPROV.	7.18%	4.17%	8.11%

Table 4.6: MSE loss values for different growing sub-steps and relative average improvement.

	Train	Validation	Test
BASE TR.	0.1448 ± 0.0065	0.1556 ± 0.0273	0.1545 ± 0.0075
REPL.	0.0799 ± 0.0059	0.1043 ± 0.0432	0.1023 ± 0.0180
REFIN.	0.0746 ± 0.0088	0.0959 ± 0.0394	0.1021 ± 0.0160
IMPROV.	48.48%	38.37%	33.9%

Architecture and hyperparams: the base network is composed of 17 hidden units, with a replacement of 1 subnet per neurons; each subnet has 7 hidden neurons, the learning rates used in the 3 steps are 0.1, 0.01, 0.001 respectively and the L2 penalties are 0.01, 0.0001, 0.0001.

Wine

Dataset: The Wine dataset is composed of 178 examples and 13 features and belonging to 3 classes. Features represent chemical data related to wine, aiming at defining its provenance. In tables 4.7 and 4.8 are reported the accuracy and loss values for training, validation and test set for each of the growing step.
Results: the performance of the model are good since the first sub-step. Anyway it happens that the growth process results in a slightly worsening of training set performances (that were very high since the beginning). On the other hand, this entails a generalization improvement, as can be seen in the validation performances. The same happens for the test set accuracy, even if it is not reflected in the test loss behaviour. At the end, the model reaches a test accuracy of 99.43%, with an improvement of 1.13% with regard to the base model.

Table 4.7: Accuracy values for different steps. and relative improvement.

	Train (%)	Validation (%)	Test (%)
BASE TR.	99.79 ± 0.36	98.21 ± 3.09	98.30 ± 0.98
REPL.	99.17 ± 0.59	100.00 ± 0.00	98.86 ± 1.14
REFIN.	99.58 ± 0.42	100.00 ± 0.00	99.43 ± 0.98
IMPROV.	-0.21%	1.79%	1.13%

Table 4.8: MSE loss values for different steps. and relative improvement.

	Train (10^{-3})	Validation (10^{-3})	Test (10^{-3})
BASE TR.	5.40 ± 1.30	11.69 ± 11.65	7.98 ± 4.26
REPL.	10.37 ± 4.54	9.23 ± 8.70	12.96 ± 6.13
REFIN.	5.37 ± 2.24	7.04 ± 6.57	9.17 ± 4.56
IMPROV.	0.56%	39.78%	-14.91%

Architecture and hyperparams: The base network is composed of 10 hidden units and 3 output units. The growing step has been realized using one subnet per neuron where each subnet had 12 hidden units. Learning rates used: 0.1, 0.01, 0.001, L2 penalties: 0.01, 0.0001, 0.0001.

Ozone

Dataset: The Ozone dataset is composed of 2536 examples, 72 features, and 2 classes. The data is relative to climate features.
Results: tables 4.9 and 4.10 report the accuracy and loss values for training, validation and test set, for each of the growing step. There is a very small improvement, but with respect to the std.dev it is not very significant.

Table 4.9: Accuracy values for different steps.

	Train (%)	Validation (%)	Test (%)
BASE TR.	97.50 ± 0.47	96.73 ± 0.23	97.24 ± 0.18
REPL.	97.28 ± 0.25	96.99 ± 0.23	97.28 ± 0.20
REFIN.	97.38 ± 0.43	96.86 ± 0.00	97.32 ± 0.27
IMPROV.	-0.18%	0.13%	0.08%

Table 4.10: MSE loss values for different steps.

	Train (10^{-2})	Validation (10^{-2})	Test (10^{-2})
BASE TR.	1.92 ± 0.23	2.42 ± 0.28	2.32 ± 0.08
REPL.	1.91 ± 0.24	2.33 ± 0.21	2.35 ± 0.20
REFIN.	1.74 ± 0.21	2.20 ± 0.17	2.30 ± 0.19
IMPROV.	9.37%	9.09 %	0.86%

Architecture and hyperparams: The base model has 40 neurons. The growing step is realized using 4 subnets, each having 25 hidden units and 10 output units. Learning rates used are 0.001, 0.1, 0.001 and L2 penalties 0.01, 0.001, 0.0001.

Pima

The dataset contains 768 examples, 8 features and 2 classes.
Results. In tables 4.11 and 4.12 loss and accuracy improvements are reported. We observe an improvement of the accuracies on validation and test set.

Table 4.11: Accuracy values for different sub-steps.

	Train (%)	Validation (%)	Test (%)
BASE TR.	78.19 ± 0.68	78.02 ± 3.31	75.00 ± 2.68
REPL.	77.27 ± 0.63	79.74 ± 3.31	77.08 ± 2.05
REFIN.	77.90 ± 0.80	80.60 ± 02.83	77.08 ± 2.05
IMPROV.	-0.29%	2.58%	2.08%

Table 4.12: MSE loss values for different sub-steps.

	Train	Validation	Test
BASE TR.	0.15144 ± 0.00327	0.13778 ± 0.02163	0.16137 ± 0.01360
REPL.	0.14928 ± 0.00301	0.13612 ± 0.01974	0.16712 ± 0.01250
REFIN.	0.14677 ± 0.00305	0.13601 ± 0.01679	0.16564 ± 0.00965
IMPROV.	3.08%	1.28%	-2.65%

Architecture: Base network has 15 hidden units, 1 subnet per neuron has been replaced, each subnet has 7 hidden units. Learning rates: 0.1, 0.001, 0.001, L2 penalties: 0.001, 0.01, 0.01.

Blood

The data is related to the blood transfusion service centre in Taiwan. This ha 748 examples and 5 features.
Tables 4.13 and 4.14 represent model performances for the single growing sub-step and the relative improvements. It is clearly visible the improvement on each data split, both from the accuracy and loss perspective.

Table 4.13: Accuracy values for different steps.

	Train (%)	Validation (%)	Test (%)
BASE TR.	77.33±1.30	76.75±3.80	77.14±0.58
REPL.	79.66±1.83	77.63±3.37	79.28±3.67
REFIN.	79.17±1.17	78.51±2.88	80.35±3.03
IMPROV.	1.84%	1.76%	3.21%

Table 4.14: MSE loss values for different steps.

	Train	Validation	Test
BASE TR.	0.16148 ± 0.00218	0.15271±0.01092	0.16343±0.00851
REPL.	0.13931±0.00712	0.13641±0.01577	0.15661±0.01045
REFIN.	0.13696±0.00450	0.13196±0.01343	0.15165±0.00866
IMPROV.	15.18%	13.58%	7.20%

Architecture and hyperparameters. The main model has 12 hidden units, and in the growing step one subnet per neuron has been used. Each subnet has 12 hidden units. Learning rates used 0.1, 0.01, 0.001, and weight decays: 0.001, 0.0001, 0.001.

Adult

Dataset. The adult dataset contains census data features, with labels indicating if the year income is lower or higher of $ 50.000. It contains 48842 example, 14 features and 2 classes. In tables 4.15 and 4.16 are reported respectively the accuracy and loss values for each growing step (and relative sub-steps). Here the growing step has been recursively applied generating a 4 hidden layer architecture. Anyway we have also reported the 4-th growing step albeit the best architecture in obtained in the 3-rd growing step. The improvement data is related to the best architecture found.

Table 4.15: Accuracy values for different growing steps.

		Train (%)	Val. (%)	Test (%)
	BASE TR.	85.67	86.09	85.38
STEP-1	REPL.	85.86	86.31	85.49
	REFIN.	86.13	86.31	85.60
STEP-2	REPL.	85.91	86.40	85.50
	REFIN.	86.37	86.18	85.53
STEP-3	REPL.	85.66	86.46	**85.66**
	REFIN.	85.66	86.28	85.63
STEP-4	REPL.	85.58	86.71	85.28
	REFIN.	85.60	86.31	85.20
	IMPROV.	-0.01	0.37	0.28

Table 4.16: MSE loss values for different growing steps.

		Train	Val.	Test
	BASE TR.	0.09824	0.09491	0.09958
STEP-1	REPL.	0.09091	0.08836	0.10071
	REFIN.	0.08836	0.08717	0.09902
STEP-2	REPL.	0.09111	0.08946	0.10115
	REFIN.	0.08817	0.08859	0.10012
STEP-3	REPL.	0.09395	0.09185	0.10339
	REFIN.	0.09391	0.09220	0.10365
STEP-4	REPL.	0.09488	0.09222	0.10557
	REFIN.	0.09525	0.09253	0.10603
	IMPROV.	4.37%	3.22%	-3.83%

Architecture and hyperparams. Base model has 8 hidden units, then replaced by 4 subnets each having 20 hidden units and 2 output units. Learning rates: 0.001, 0.01, 0.001 and L2 penalties: 0.01, 0.0001, 0.0001.

Comparison

This section states how the DGNA experimental results stands with respect to other machine learning models. Here we compare the results obtained with our model, introduced in previous subsections, with the ones obtained using other state of the art deep neural models.

As previously mentioned, in this experimental section we rely on the UCI dataset organization generated in [20], where they defined folds splitting (both for hyperparameters search and crossvalidation procedure) for 121 UCI datasets. This dataset collection is becoming more and more a valid alternative to benchmarking general purpose neural network models, especially for the very different nature of the datasets at hand, and their different characteristics (both in terms of number of features and available examples). This can lead towards a fairer models evaluation.
Our results are compared with the ones obtained in [34]. Although the work done in [34] is not related to the topic of neural architecture search or growing networks, in order to assess the performances of their proposed solutions, they trained a significant number of neural models on all of the 121 UCI dataset, using the same data organization defined in [20]. For each of the 121 dataset, they trained 7 different type of neural networks, and for each network an optimization was performed in terms of architecture (number of layers and neurons per layer) and hyperparameters, using an ad-hoc validation set. The obtained architectures and hyperparameters were then used to train and evaluate the models on the pre-defined 4 folds. The considered hyperparameters are reported in table 4.17. We mainly choose this benchmark [34] because of the significant number of results available for comparison and secondly because they used exactly the same fold partitions.
At the end, the DGNA results are compared with self-normalizing neural networks (SNN) [34], ReLU networks [24], residual networks (ResNet) [27], networks with batch-normalization (BN) [30], network with weight normalization (WN) [50] and network with layer normalization (LN) [2].
Results of the comparison are in table 4.18. Furthermore, although not explicitly reported, authors in [34] state that SNN architectures have an average of 10.8 layers, BN 6.0, WN 3.8, LN 7.0 ReLU 7.1, and 6.35 blocks for ResNet, that can be considered way bigger architectures if compared to the ones obtained with our algorithm.

In table 4.18 we compare the results obtained with out proposed DGNA, with results of different neural models obtained in [34]. We need to bear in mind that this results are strictly related to the particular dataset organization used [20], and may result slightly different from ones obtained with a generic k-folds crossvalidation procedure. Unfortunately standard deviations are not reported in [34].
Beyond noting that all considered results are in line, it is interesting to note that in

Table 4.17: Considered hyperparameters

Hyperparameter	Considered values
# hidden units	{256, 512, 1024}
# hidden layers	{2, 3, 4, 8, 16, 32}
learning rate	{1, 0.1, 0.01}
dropout rate	{0, 0.5}
layer form	{rectangular, conic}

Table 4.18: Accuracy comparison for different models trained with the UCI datasets

Dataset	DGNA	SNN	ReLU	ResNet	BN	WN	LN
Ionoshpere	93.47$\pm$0.49	88.64	90.91	**95.45**	94.32	93.18	94.32
Wine	**99.43**$\pm$0.98	97.73	93.18	97.73	97.73	97.73	97.73
Vertebral	**87.01**$\pm$1.84	83.12	87.01	83.12	83.12	66.23	84.42
Blood	**80.35**$\pm$3.03	77.01	77.54	80.21	76.47	75.94	71.12
Pima	**77.08**$\pm$2.05	75.52	76.56	71.35	71.88	69.79	69.79
Ozone	97.32$\pm$0.27	97.00	97.32	96.69	96.69	**97.48**	97.16
Adult	**85.66**	84.76	84.87	84.84	84.99	84.53	85.17

most of the cases, results obtained with our model surpass the ones obtained with other deep neural model, in particular this happens in 4 of the 7 use cases, while with the Vertebral dataset we exactly reach the top performance 87.01% of accuracy, also reached with a deep ReLU network. In the case of Ionosphere dataset, DGNA reaches the 3-rd best results and with Ozone dataset we reach the second best results.

Results considerations

In this experimental section we run our proposed model, the DGNA, on 7 UCI datasets. In each single experiment, an improvement has been recognized for each of the growing sub-steps (base training, replacement and refinement). This let us assume that the proposed growing algorithm is effective.
Each of the single substep contributes to improve the model loss and consequently of the relative accuracy. Furthermore, obtained results are comparable with performances reached using well established deep neural network models, and in few cases are even better.
At the end of the day, our model consists in a different way of defining multilayer architecture, by progressively adding computational units, where this process is driven by the well defined goal of progressively define more complex decision regions. In a certain way, what we expected, and that is partially confirmed by experimental results, is that the application of a growing-step, at least results in a network having the same performance of the previous architecture. It can be clearly seen in the Adult dataset results, table 4.15 and table 4.16. Although actually a theoretical proof is missing, relying on the same intuition that lead to the definition of this new growing algorithm, this is due to the particular growing strategy chosen. Basically because a subnet used to replace a neuron with its input weights, is obviously able to realize at least the old straight line separation surface previously defined by the latter neuron configuration. Anyway, by adding further and further layers, with the consequent estimation of subnet's target, numerical problem are clearly introduced, potentially taking to a gradually worsening of the model performances.
Another point to consider is the latent role of "target propagation". Beyond the main goal of defining targets for the inner subnets, this technique was also conceived with the aim of simplifying a certain given problem. So what practically happens is that the learning problem that the subnet has to deal with, is typically a simpler learning problem than the original one. Also here, a theoretical proof would be helpful. While this idea was the same also for the *Depth Growing Neural Network* framework, this could not happen because of the particular configuration of subnet architecture; and propagating targets did not simplify learning problems as expected. These two mentioned points are the main differences of the *Downward-Growing Neural Architecture* with respect to other growing models, and it seems to let the model achieve promising results.

4.5 Remarks

In this chapter we presented the solutions engineered to solve some limitations of the DGNN framework. These solutions resulted in the definition of a brand new framework, the *Downward-Growing Neural Architecture* (DGNA). The latter basically differs from the DGNN in the growing strategy adopted and the learning algorithm. While in DGNN only the hidden units are replaced (recursively), without altering the remaining part of the architecture, here in the DGNA the subnet replace the hidden neurons and their input connections too, by adding computational units at the bottom of the whole model. Defining in this way exactly a downward growing architecture.
Then an experimental setup has been realized, testing the model with 7 datasets of the UCI repository [17].
Experimental results seem to assess that the algorithm works as expected, continually improving the model performance after the single growing steps (and substeps). An experimental comparison of the obtained results has been done with other deep learning model, and in some cases our results outperform all the others.

Chapter 5

Conclusions

The search for the neural network architecture that best fits the learning problem is still an open problem. Nowadays, although some alternative exists, most practitioners still rely on trial and error approaches. In this Thesis we tackled the above mentioned problem, trying to propose brand new solutions.
The efforts done in reaching the goal, entailed the definition of three milestones, that reflect the organization of this work: *Target Propagation*, *Depth-Growing Neural Network* and *Downward-Growing Neural Architectures*.
Since the beginning, the idea was to define a growing neural network model by progressively adding more complex inner components. First attempts were in the direction of defining working *target propagation* techniques, in order to define inner subnet-specific datasets to properly train the new introduced components. In chapter 2 is discussed the idea of target propagation, and new algorithms are proposed. Then an application of the target propagation paradigm as a refinement learning tool is tested.

The first working version of the growing model that implements our aimed solution is introduced in chapter 3, the *Depth Growing Neural Network* framework. It aims at autonomously find the best architecture while at the same time keep updating its parameters configuration. This is reached by evaluating the performances of the internal computational units, the metaneurons, based on which few of them, the worst performing ones, are chosen to be grown. As said in previous sections, this goal has been reached thanks to the definition of ad-hoc *target propagation* techniques. From the experimental section emerges that the framework is only partially effective. Albeit it does not always improve accuracy statistics in classification problems, it significantly reduces the loss function values.
Aware of the limitations intrinsic of the DGNN model, further research efforts led to the definition of the *Downward-Growing Neural Architecture* framework. The key ideas and the learning dynamic are extensively discussed in chapter 4. Here an

experimental section was carried on 7 datasets of the UCI repository, and a comparison with other results obtained using different deep neural architectures is held. From the experimental section emerges that the application of the DGNA growing algorithm significantly improves the model performances on each of the 7 dataset; furthermore the comparison shows that in 4 of the 7 tested dataset our results surpass all the others.

The conducted research led to definition of a robust growing model, with promising performances. Few pieces are still missing to complete the puzzle, that could be helpful in understanding some of the model properties. First, a theoretical proof that the adopted learning strategy helps the learning convergence of the model would be an interesting success. Furthermore, smarter techniques to drive the selection of the architecture of the subnets would be very helpful.
Another direction that has not been pursued is that of realizing an asynchronous growing strategy, were not all the neurons are replaced at the same time, but the operation is done on demand. A simple version of this asynchronous growing strategy was implemented in the DGNN. A solution may be to prune few nodes of the network (and relative connection weights) after each growing step, in this way only most important units are kept, or further grown. Eventually, another evolution to consider may be the extension of the framework to deal with sequential data. In this context, the adaptive structure of the model could play a major role in processing variable-length sequences.
The philosophy that drove the development of the framework, places it in a direction that seems to be the natural evolution of learning paradigms that are naturally inspired, like artificial neural networks are.

Appendix A

Publications

Conference/Workshop papers

1. Marco Bongini, **Vincenzo Laveglia**, Edmondo Trentin, “A Hybrid Recurrent Neural Network / Dynamic Probabilistic Graphical Model Predictor of the Disulfide Bonding State of Cysteines from the Primary Structure of Proteins”. *ANNPR*, pages:257–268, 2016. **Candidate’s contributions**: data preprocessing algorithms evaluation, experimental setting design and implementation, analysis of results.

2. **Vincenzo Laveglia**, Edmondo Trentin, “ A Refinement Algorithm for Deep Learning via Error-Driven Propagation of Target Outputs”. *ANNPR*, pages:79–89, 2018. **Candidate’s contributions**: target propagation algorithm ideation, experimental setting design and implementation, analysis of results.

Journal papers

1. Marco Bongini, **Vincenzo Laveglia**, Edmondo Trentin, “Dynamic Hybrid Random Fields for the Probabilistic Graphical Modeling of Sequential Data: Definitions, Algorithms, and an Application to Bioinformatics.”. *Neural Processing Letters 48(2)*, pages:733–768, 2018. **Candidate’s contributions**: ideation and design of a postprocessing module, experimental setting design and implementation, analysis of results.

Bibliography

[1] Forest Agostinelli, Matthew D. Hoffman, Peter J. Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks. *CoRR*, abs/1412.6830, 2014.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[4] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 153–160, 2006.

[5] Yoshua Bengio, Dong-Hyun Lee, Jörg Bornschein, and Zhouhan Lin. Towards biologically plausible deep learning. *CoRR*, abs/1502.04156, 2015.

[6] Yoshua Bengio, Patrice Y. Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[8] Ilaria Castelli and Edmondo Trentin. Semi-unsupervised weighted maximum-likelihood estimation of joint densities for the co-training of adaptive activation functions. In *Partially Supervised Learning - First IAPR TC3 Workshop, PSL 2011, Ulm, Germany, September 15-16, 2011, Revised Selected Papers*, pages 62–71, 2011.

[9] Ilaria Castelli and Edmondo Trentin. Supervised and unsupervised co-training of adaptive activation functions in neural nets. In *Partially Supervised Learning - First IAPR TC3 Workshop, PSL 2011, Ulm, Germany, September 15-16, 2011, Revised Selected Papers*, pages 52–61, 2011.

[10] Ilaria Castelli and Edmondo Trentin. Combination of supervised and unsupervised learning for training the activation functions of neural networks. *Pattern Recognition Letters*, 37:178–191, 2014.

[11] Chyi-Tsong Chen and Wei-Der Chang. A feedforward neural network with function shape autotuning. *Neural Networks*, 9(4):627–641, 1996.

[12] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[13] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 874–883, 2017.

[14] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 874–883, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

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

[16] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.

[17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[19] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in neural information processing systems*, pages 524–532, 1990.

[20] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[21] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*,

pages 1165–1173, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[22] Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in neural information processing systems*, pages 625–632, 1995.

[23] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pages 249–256, 2010.

[24] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, pages 315–323, 2011.

[25] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2672–2680, 2014.

[26] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1319–1327, 2013.

[27] 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.

[28] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Automated machine learning-methods, systems, challenges, 2019.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org, 2015.

[31] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 1627–1635, 2017.

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

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[34] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[35] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[36] Vincenzo Laveglia and Edmondo Trentin. A refinement algorithm for deep learning via error-driven propagation of target outputs. In *Artificial Neural Networks in Pattern Recognition - 8th IAPR TC3 Workshop, ANNPR 2018, Siena, Italy, September 19-21, 2018, Proceedings*, pages 78–89, 2018.

[37] Yann Le Cun. Learning process in an asymmetric threshold network. In E. Bienenstock, F. Fogelman Soulié, and G. Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 233–240, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[38] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[39] Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part I*, pages 498–515, 2015.

[40] Thomas Martinetz. Competitive hebbian learning rule forms perfectly topology preserving maps. In *International conference on artificial neural networks*, pages 427–434. Springer, 1993.

[41] Thomas Martinetz, Klaus Schulten, et al. A" neural-gas" network learns topologies. 1991.

[42] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[43] Marvin L Minsky and Seymour Papert. Perceptrons: an introduction to computational geometry. 1969.

[44] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

[45] Quynh Nguyen, Mahesh Chandra Mukkamala, and Matthias Hein. Neural networks should be wide enough to learn disconnected decision regions. In *International Conference on Machine Learning*, pages 3737–3746, 2018.

[46] Matthew Olson, Abraham Wyner, and Richard Berk. Modern neural networks generalize on small data sets. In *Advances in Neural Information Processing Systems*, pages 3619–3628, 2018.

[47] R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3):406–413, 1955.

[48] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[49] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[50] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

[51] Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *null*, page 958. IEEE, 2003.

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

[53] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[54] Edmondo Trentin. Networks with trainable amplitude of activation functions. *Neural Networks*, 14(4-5):471–493, 2001.

[55] Guangcong Wang, Xiaohua Xie, Jianhuang Lai, and Jiaxuan Zhuo. Deep growing learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2812–2820, 2017.

[56] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

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

[58] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. In *International Conference on Learning Representations*, 2018.