entropy

MDPI

Article

Downward-Growing Neural Networks

Vincenzo Laveglia [1,†,‡] and Edmondo Trentin [2,*,‡]

[1] DINFO, Università di Firenze, Via di S. Marta 3, 50139 Firenze, Italy
[2] DIISM, Università di Siena, Via Roma 56, 53100 Siena, Italy
* Correspondence: trentin@dii.unisi.it; Tel.: +39-0577-234636
† Current address: Consorzio Interuniversitario di Risonanze Magnetiche di Metallo Proteine, Via Luigi Sacconi 6, 50019 Sesto Fiorentino, Italy.
‡ These authors contributed equally to this work.

Abstract: A major issue in the application of deep learning is the definition of a proper architecture for the learning machine at hand, in such a way that the model is neither excessively large (which results in overfitting the training data) nor too small (which limits the learning and modeling capabilities of the automatic learner). Facing this issue boosted the development of algorithms for automatically growing and pruning the architectures as part of the learning process. The paper introduces a novel approach to growing the architecture of deep neural networks, called downward-growing neural network (DGNN). The approach can be applied to arbitrary feed-forward deep neural networks. Groups of neurons that negatively affect the performance of the network are selected and grown with the aim of improving the learning and generalization capabilities of the resulting machine. The growing process is realized via replacement of these groups of neurons with sub-networks that are trained relying on ad hoc target propagation techniques. In so doing, the growth process takes place simultaneously in both the depth and width of the DGNN architecture. We assess empirically the effectiveness of the DGNN on several UCI datasets, where the DGNN significantly improves the average accuracy over a range of established deep neural network approaches and over two popular growing algorithms, namely, the AdaNet and the cascade correlation neural network.

Keywords: deep neural network; deep learning; adaptive architecture; growing neural network; target propagation

Citation: Laveglia, V.; Trentin, E. Downward-Growing Neural Networks. *Entropy* **2023**, *25*, 733. https://doi.org/10.3390/e25050733

Academic Editor: Gholamreza Anbarjafari

Received: 28 February 2023
Revised: 8 April 2023
Accepted: 24 April 2023
Published: 28 April 2023

1. Introduction

Selecting the best architecture for a given learning task has always been an open issue in the training of deep neural networks (DNNs). Trial-and-errors and heuristic solutions still represent the state of the art. Therefore, the selection process is frustrating and heavily time-consuming, and it generally ends up in sub-optimal architectures.

Formally, given a learning task $\mathcal{T}$ represented in terms of the corresponding supervised training set $\mathcal{D} = \{(\mathbf{x}_j, \mathbf{y}_j)_{j=1}^{N}\}$, the following steps are required in the development of a feed-forward DNN capable of tackling $\mathcal{T}$: (1) set the number of input and output units equal to the corresponding dimensionalities of the input and target outputs in $\mathcal{D}$, respectively; (2) fix the form of the activation functions in the output layer, such that their codomain matches the range of the target values in $\mathcal{D}$; (3) most crucially, fix the internal structure of the DNN (which utterly affects the computational capabilities of the model). This includes fixing the number of hidden layers, the size of each such layer, and the form of the corresponding activation functions. If dealing with many hidden layers, quasi-linear activation functions are advisable in order to overcome numerical issues such as vanishing gradients [1]. Once the architecture has been fixed, a proper learning strategy shall be implemented. This strategy involves the selection of other hyperparameters, including the mini-batch size, a drop-out value [2], the batch-normalization [3], etc. All in all, model

selection is no straightforward process, and it typically involves (implicitly or explicitly) applying a computationally expensive search strategy.

This paper investigates a novel framework for the automatic completion of the aforementioned steps (2) and (3), that is, the data-driven adaptation of the DNN architecture and, implicitly, of the corresponding neuron-specific activation functions. The proposed framework is herein referred to as the downward-growing neural network (DGNN). The learning process in the DGNN unfolds deep architectures by means of local, piecewise training sub-processes in an incremental way, with no need for the usual overall backpropagation of partial derivatives throughout the whole DNN. Inherently, this can be seen as an instance of the divide-and-conquer strategy. Section 4 makes it explicit that the DGNN realizes a model of exploratory causal analysis capable of causal discovery [4,5].

Before presenting the DGNN, we begin the treatment by studying a motivating example (Section 1.1) and surveying the literature related to the present research (Section 1.2). The details of the proposed algorithm are presented in Section 2 ("Materials and Methods"). The latter introduces the first two novel target-propagation methods (Section 2.1), namely, the residual driven target propagation (Section 2.1.1) and the gradient-based target propagation (Section 2.1.2), used as building blocks for the DGNN growing and learning procedure (Section 2.2). The outcome of experiments conducted on datasets drawn from the UCI repository is reported in Section 3 ("Results"), where the DGNN is compared favorably with established DNN paradigms and growing algorithms. Section 4 ("Conclusions") draws the concluding remarks and pinpoints major directions for future research work.

1.1. Motivating Example

Let us consider a feed-forward neural network, for instance, a multilayer perceptron (MLP). To fix ideas, assume that the MLP has three layers, namely, L_0, L_1, and L_2, where L_0 (the input layer) has d input units (d being the dimensionality of the feature space), the hidden layer L_1 has arbitrary size, and L_2 (the output layer) of 1-dimensional (i.e., there is only one output unit). 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, that is, a mapping $R^{d_{i-1}} \rightarrow R^{d_i}$ where d_i is the i-th layer dimensionality (i.e., the corresponding number of units), and $\sigma : R \rightarrow R$ is the usual element-wise activation function. The function f_0 is associated to the input layer and, as such, is not considered for all practical intents and purposes (in fact, input signals do not undergo any transformation before being propagated forward through the network).

Let us consider a classification problem with two classes, say, ω_1 and ω_2, and let $T = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j)_{j=1}^N,\ \mathbf{x}_j \in R^2, \hat{\mathbf{y}}_j \in \{0, 1\}\}$ be a supervised dataset where $\hat{\mathbf{y}}_j = 0$ if $\mathbf{x}_j$ is in class ω_1 and $\hat{\mathbf{y}}_j = 1$ else. We write $\hat{\mathbf{y}}$ to represent the network output when the network is fed with the generic input $\mathbf{x}$. Finally, to fix ideas, we assume that a logistic sigmoid $\sigma(.)$ is associated to the output neurons of the network.

Once the training has been completed, we can consider the network weights as constants. The k-th neuron in L_1 realizes the logistic regression $o_k = \sigma(x_1 w_{k1} + x_2 w_{k2} + b_k)$ which, in turn, realizes the k-th component of the layer-specific multi-dimensional function $f_1(.)$. Depending on $\mathbf{x}$, o_k is valued along the tails of $\sigma(.)$ or in its middle range. Whenever the module of the connection weights is large, o_k is valued along the tails of the sigmoid for most of the inputs $\mathbf{x}$; that is to say, o_k turns out to be close to either 0 or 1.

We write R_0 to represent the decision region for class ω_1, that is, the subspace of $\mathbb{R}^2$ where o_k is below the decision threshold (namely, $o_k \leq \frac{1}{2}$), and R_1 to represent the decision region for class ω_2 (i.e., the subspace of $\mathbb{R}^2$ where $where o_k > \frac{1}{2}$). The set of points where $o_k = \frac{1}{2}$ forms the inter-class separation surface. 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 $F_i(\mathbf{x}) = f_i(f_{i-1} \ldots f_0(\mathbf{x}))$ and, in the specific case of sigmoid activation functions, $\gamma = \frac{1}{2}$. For sigmoids, a generic $\mathbf{x} = (x_1, x_2)$ lies on the separation surface when $x_1 w_{k1} + x_2 w_{k2} + b_k = 0$, that can be easily rewritten as

$$x_2 = -x_1 \frac{w_{k2}}{w_{k1}} - \frac{b_k}{w_{k1}} \tag{1}$$

that is, the equation of a line having slope $-\frac{w_{k2}}{w_{k1}}$ and offset $-\frac{b_k}{w_{k1}}$. The computation realized by layer L_1 of the MLP is the set of the outputs of the neuron-specific logistic sigmoids for that layer. It is straightforward to see that this pinpoints which one of the two neuron-specific decision regions (R_0 or R_1) the generic input $\mathbf{x}$ belongs to, for each and every one of the individual neurons. A graphical representation is shown in Figure 1.

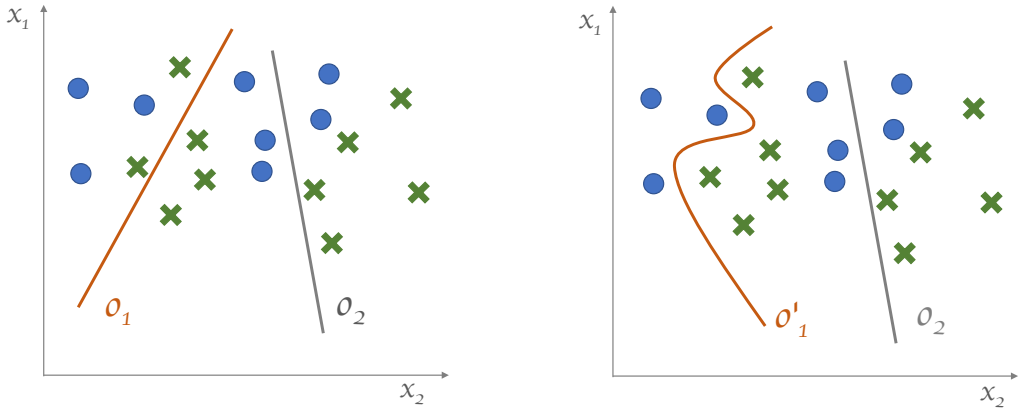

Figure 1. Data in $\mathbb{R}^2$ belong to two classes, represented by crosses and circles. (**Left**): separation surfaces defined by two hidden neurons. (**Right**): separation surfaces expected to be generated by a growing model, where o'_1 is the grown version of o_1.

The argument we just used for L_1 can be extended to L_2, as well. The difference is that L_2 is fed with the outputs of L_1. Furthermore, L_2 being the output layer, its separation surface corresponds to the separation surface of the classifier. In general, we can say that the overall separation surface $\mathcal{S}_i^k$ realized by the network is a function of the separation surfaces defined at the previous layer: $\mathcal{S}_{i-1}^1, \mathcal{S}_{i-1}^2, \ldots, \mathcal{S}_{i-1}^{d_{i-1}}$. In short, the output separation surface is $\mathcal{S}_2^k = \phi(\mathcal{S}_1^1, \ldots, \mathcal{S}_1^{d_1})$. In particular, in the simple architecture at hand, it is seen that $\mathcal{S}_2^k$ is approximately in a piecewise linear form, where each linear segment corresponds to the separation surface generated in the preceding layer. It turns out that the overall separation surface is built from the linear segments realized by the neurons in L_1, as shown in Figure 1 (left).

In most of the cases, in practice, quasi-piecewise linear decision regions are not sufficient to separate data effectively. In this scenario, what we expect from a growing model is the capability to overcome such limitations by switching from piecewise-linear to generic decision surfaces. Furthermore, we want the model to define decision regions that can adapt to the model needs (i.e., to the nature of the data $\mathcal{D}$ and of the specific learning task $\mathcal{T}$) in order to improve the performance of the resulting machine. This behavior is shown graphically in Figure 1 (right).

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

1.2. Related Works

The idea of learning/evolving the architecture of a neural network is not, per se, new. Early attempts date back to the late 1980s [6]. In this section, we review major, popular techniques for growing neural network architecture that relate somehow to the approach presented in this paper.

One of the most popular and effective paradigms for growing neural architectures is the cascade-correlation neural network [7]. It prescribes initializing the network as a

minimal architecture with no hidden layers. After a first learning stage (based on gradient-descent), an iterative growing procedure is applied. It consists of adding a single neuron per time to the architecture. New forward connections are created, linking each pre-existing neuron (except for the output units) to the newly added neuron. The corresponding connection weights are learned by maximizing the correlation between the outcome of the new neuron and the output of the network. Afterwards, the values of these connection weights are clamped, and the output weights are learned via plain gradient-descent. Each new neuron partakes, in turn, in feeding the input to the next neuron to be added. This algorithm generates a particular instance of a deep architecture, where each hidden layer is composed of a single neuron and each internal neuron is fed from all the previous neurons (either input or hidden neurons).

An unsupervised incremental architecture-growing approach is represented by the growing neural gas [8], an extension of the traditional neural gas model [9]. A growing algorithm for semi-supervised learning is presented in [10]. As learning proceeds, more and more computational power is required of the learning machine to capture the input-to-output relationship encapsulated within the ever-increasing labeled fraction of the dataset. In [10], this is accomplished by creating new, additional layers that are plugged into the network architecture. Any such new layer is initialized as a replica of the previous one; then, a fine-tuning procedure is applied to optimize the parameters stemming from the introduction of the new layer.

A recent trend occurred in the development of algorithms capable of realizing dynamic architectures (including growing architectures) Recently, adaptive (e.g., growing) architectures proved suitable to continual learning setups (i.e., setups where the learning task changes over time). In particular, the approach presented in [11] exploits the knowledge encapsulated within a previously trained machine in order to train a new, larger neural architecture capable of modeling the new instances of the time-dependent learning task at hand.

A significant approach that relates to the present research is the AdaNet [12]. The AdaNet is initialized with a simple base-architecture, and new neurons are progressively added for as long as the performance improves. The criterion function to be minimized involves an architecture-driven regularized empirical risk where the complexity of the architecture plays the role of the regularization term. The growing stage goes as follows: given the base-architecture $\mathbf{h}_\ell$ having ℓ layers, two candidate networks $\mathbf{h}'_\ell$ and $\mathbf{h}'_{\ell+1}$, are generated, having ℓ and $\ell+1$ layers, respectively. The generic $k+1$-th layer in both candidate nets is fed with the output of the k-th layer in $\mathbf{h}_\ell$, leveraging the embeddings of the data that $\mathbf{h}_\ell$ learned already. Then, the candidate models undergo completion of their training process. Eventually, a new base-architecture is selected between $\mathbf{h}'_\ell$ and $\mathbf{h}'_{\ell+1}$ based on the corresponding performance in terms of the given loss function, and the whole procedure is iterated. It is seen that the computational cost of this growing algorithm can be very high, especially when the learning process ends up in large architectures.

Splitting steepest descent for growing neural architectures [13] is a technique that "splits" an individual neuron by replacing it with a set of new neurons whenever the learning process cannot improve the loss any further. Any such set of new neurons is initialized in such a way that the sum of their outputs equals the output of the neuron that underwent splitting. An ad hoc metric is defined for choosing the next neuron to be split. The approach was extended to the so-called firefly neural architecture descent in [14]. The latter sides the neuron-splitting mechanism with other growing tools that allow for the modification in width and depth of the neural architecture at hand.

Another approach called Gradmax was recently introduced in [15]. It focuses on an initialization procedure for the ingoing and outgoing connection weights of new neurons that have been introduced in the architecture at hand. The technique revolves around the idea of initializing the weights by solving an optimization problem such that (1) the output of the network is not initially affected by the activity of the new neurons, and (2) the

gradient of the loss function with respect to the new weights is maximum in order to speed up the learning process.

2. Materials and Methods

In light of the motivating example analyzed in Section 1.1, and relying on the notation introduced therein, we hereafter extend our scope to any generic supervised learning task. As we have seen, in a two-class classification task, the linear decision surface $\mathcal{S}_i^k$ is the subset of the feature space where o_k is equal to a certain value γ. In particular, in the case of sigmoid activation function, we have $\gamma = \frac{1}{2}$, and $o_k = \sigma(\mathbf{w}_k^{in}\mathbf{x} + b_k)$. The latter depends on the neuron input weights $\mathbf{w}_k^{in} = (w_{k1}, w_{k2})$ and on the bias. We replace the neuron and all its input connections $(\mathbf{w}_k^{in}, b_k)$ with a more general, adaptive processing component realizing a nonlinear activation function $\varphi : \mathbb{R}^{d_0} \rightarrow \mathbb{R}$ such that the corresponding decision surface $\mathcal{S}^k$ results in a more flexible adaptive form. Such an adaptive processing component is realized via a smaller neural network *Sub* that we call subnet. Therefore, $o_k = \varphi(\mathbf{x}; \mathcal{W})$ where $\mathcal{W}$ represents the set of all the weights of the subnet. In so doing, a modification of the original network architecture is achieved. Figure 2 shows a simple yet illustrative graphical example. Although the approach has been introduced referring to the illustrative setup outlined in Section 1.1 (two-class classification task over a two-dimensional feature space), it is seen that it can be applied in a straightforward manner, as is, to generic tasks having arbitrary input dimensionalities. The procedure can be repeated multiple times recursively, leading to a progressive growth of a DNN architecture with an arbitrary number of layers. Each application of the procedure replaces either (1) a group of the bottom-most hidden neurons in the original architecture with a subnet, or (2) a group of the bottom-most hidden neurons in a subnet with a sub-subnet, and so forth in a downward-growing manner. The following sections present the algorithmic building blocks used for realizing the DGNN growing and learning processes. These building blocks are in the form of techniques for propagating target outputs to hidden neurons within the DGNN, possibly located deep down in the network (Section 2.1), as well as in the form of procedures for generating and training the subnets involved (Section 2.2). Unfamiliar readers may find a gentle introduction to the basic notions of target propagation, architecture growing, and DNN refinement in [16].

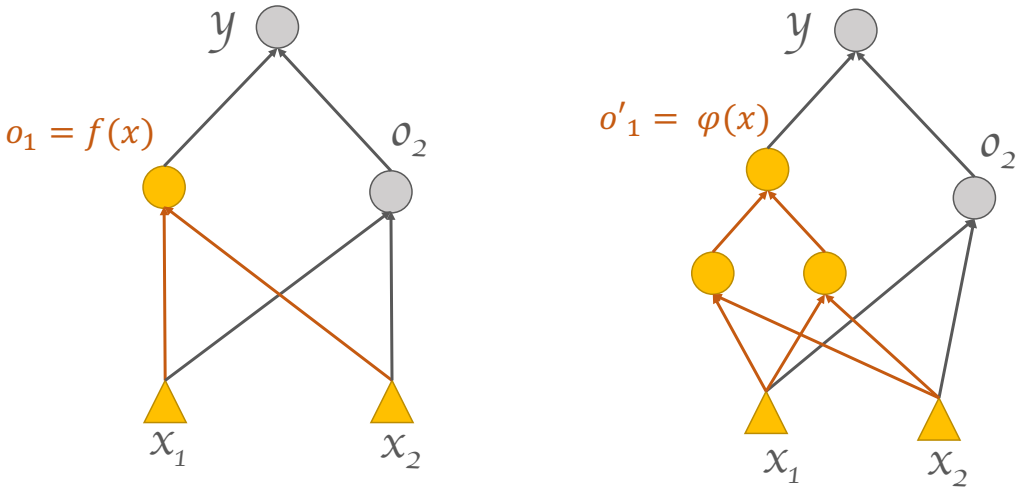

Figure 2. (**Left**): standard 1 hidden layer feed-forward network, also known as the base network. (**Right**): the grown network, after replacing the leftmost hidden neuron and its input connections with a subnet.

2.1. Target Propagation

Let us first consider the regular backpropagation (BP) algorithm [17]. The core idea behind BP is that each weight of the network is partially accountable for the output error yielded by the model when fed with any labeled example $(\mathbf{x}, \hat{\mathbf{y}})$ in the training set, where

$\hat{\mathbf{y}}$ is the target output over input $\mathbf{x}$. Let w_i be any generic connection weight in the DNN. Gradient-descent is applied in order to search for values of w_i that reduce the output error. Therefore, w_i is updated to its new value w_i' as follows:

$$w_i' = w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i} \tag{2}$$

where $\mathcal{L}$ is the loss function and η is the learning rate. In spite of its popularity and relevance, BP suffers from shortcomings when applied to deep architectures [1]. In particular, the backpropagated gradients tend to vanish in the lower layers of deep networks, hindering the overall learning process. A viable workaround was proposed in the form of Target Propagation (TP) [18,19], still an under-investigated research area. Originally proposed in [20,21] within the broader framework of learning the form of the activation functions, the idea underlying TP goes as follows. In plain BP, the signals to be backpropagated are related to the partial derivatives of the global loss function with respect to the layer-specific parameters of the DNN. To the contrary, in TP, the real target outputs (naturally defined at the output layer in regular supervised learning) are propagated downward through the DNN, from the topmost to the bottom-most layers of the network. In so doing, each layer gets explicit target output vectors that, in turn, define layer-specific loss functions that can be minimized locally without involving explicitly the partial derivatives of the overall loss function defined at the whole network level. As a consequence, the learning process is not affected by the numerical problems determined by repeatedly backpropagating partial derivatives throughout the DNN. In the TP scheme proposed hereafter, the targets are first computed for the topmost layer. Such output targets are then used for determining new targets for the DNN internal layers, according to novel downward-propagation techniques.

Given a DNN $\mathcal{N}$ having ℓ layers, let $\hat{\mathbf{y}}_\ell$ represent the generic target output of the network, which is associated to the ℓ-th layer (the output layer). The aim of TP is the computation of a proper target value $\hat{\mathbf{y}}_{\ell-1}$ for layer $\ell-1$ and, in turn, for layers $\ell-2, \ell-4, \ldots$ In order to accomplish the task, a specific function $\phi(.)$ has to be realized, such that

$$\hat{\mathbf{y}}_{\ell-1} = \phi(\hat{\mathbf{y}}_\ell) \tag{3}$$

When $\mathcal{N}$ is fed with an input vector $\mathbf{x}$, the i-th layer of $\mathcal{N}$ (for $i = 1, \ldots, \ell$, 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. For notational convenience, the layer index will be omitted when it is not needed. Hereafter, we assume that the DNN at hand is based on activation functions that are in the usual form of logistic sigmoids. Nevertheless, the following results still hold for any kind of differentiable activation functions (upon minimal adjustments of the formalization). Let us consider a supervised training set $\mathcal{D} = \{(\mathbf{x}_j, \hat{\mathbf{y}}_j) | j = 1, \ldots, N\}$. 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 $\mathcal{N}$ 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 of $\mathcal{N}$ can be written as

$$f_i(\mathbf{h}_{i-1}) = \sigma(W_i \mathbf{h}_{i-1} + \mathbf{b}_i) \tag{4}$$

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

$$F_i(\mathbf{x}) = f_i(f_{i-1}...(f_1(\mathbf{x}))) \tag{5}$$

Eventually, the function realized by $\mathcal{N}$ (which is an ℓ-layer network) is $F_\ell(\mathbf{x})$. Bearing in mind the definition of $\mathcal{D}$, the goal of training $\mathcal{N}$ is having $F_\ell(\mathbf{x}_j) \simeq \hat{\mathbf{y}}_j$ for $j = 1, ..., N$.

This is achieved by minimizing a point-wise loss function measured at the output layer. In the literature, this loss is usually the squared error, defined 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 $\mathcal{N}$ and $\|\cdot\|_2$ is the Euclidean norm. Differently from the traditional supervised learning framework for DNNs, where the targets are defined only for the neurons of the output layer, TP consists 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 DNN as well. Eventually, gradient descent with no BP may be applied in order to learn the layer-specific parameters as a function of the corresponding targets. TP is at the core of growing and training the DGNN. Two TP algorithms are proposed in the next sections, namely, residual driven target propagation (RDTP) and gradient-based target propagation. The former applies to DNNs having a single output unit (e.g., binary classifiers), while the latter is suitable to arbitrary architectures.

2.1.1. Residual Driven Target Propagation

Instead of attempting a direct estimation of the targets $\hat{\mathbf{h}}_{\ell-1}$, hereafter we aim at estimating the *residual* values $\mathbf{z}_{\ell-1}$ defined as the difference between the actual state $\mathbf{h}_{\ell-1}$ and the desired, unknown target $\hat{\mathbf{h}}_{\ell-1}$, such that $\mathbf{h}_{\ell-1} + \mathbf{z}_{\ell-1} = \hat{\mathbf{h}}_{\ell-1}$. The rationale behind using residuals is twofold:

1. Assume the network at hand, trained via plain BP over $\mathcal{D}$, converges to the global minimum of the loss function. Under the circumstances, we would just let $\hat{\mathbf{h}}_i = \mathbf{h}_i$ such that $\mathbf{z}_{\ell-1} = \hat{\mathbf{h}}_{\ell-1} - \mathbf{h}_{\ell-1}$, for $i = 1, \ldots, \ell$. To the contrary, residuals would not be null during the training: in particular, their module would start from a certain (large, in general) initial value and progressively converge to zero as training completes. Let t represent a certain training iteration, and let τ denote a certain number of consecutive training epochs. If the loss function decreases monotonically for $t = 1, 2, \ldots$, it is immediately seen that $\mathbf{z}_{\ell-1}(t+\tau) \leq \mathbf{z}_{\ell-1}(t)$. Therefore, it is seen that after pre-training the network we have $|\mathbf{z}_{\ell-1}| \ll |\mathbf{h}_{\ell-1}|$, i.e., a smaller range of the inversion function $\phi(.)$, entailing a more error-robust target propagation technique.
2. In RDTP (as we will see shortly), whenever the network evaluated over a given input pattern results in a null error, then $\mathbf{z}_{\ell-1} = \mathbf{0}$ and $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{0}$; that is, the target reduces to the actual state. In so doing, the layer-wise training steps will not entail forgetting the knowledge learned by the DNN during the preceding pre-training process. This is not guaranteed by the established target propagation techniques.

Let us stick with the single-output network case for the time being. The core of RDTP lies in the estimation of the residues $\mathbf{z}_{\ell-1}$ in the hidden layer $\ell - 1$, given the network output error $(\hat{\mathbf{y}} - \mathbf{y})^2$. Once the residues are estimated, we define the target values for layer $\ell - 1$ as $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{z}_{\ell-1}$. Note that, for notational convenience, we omitted writing explicitly the dependence of the quantities on the input pattern. Relying on the notation introduced in Section 2.1, in the present scenario ℓ and $\ell - 1$ represent the output and the hidden layer of the DNN, respectively. Let us assume that a certain value of $\mathcal{N}$ is given. Using apexes and subscripts in order to point out the layer-specific and the neuron-specific indexes, respectively, the DNN output can be written as:

$$y = \sigma_\ell\left(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} h_u^{(\ell-1)} + b^{(\ell)}\right) \tag{6}$$

with

$$h_u^{(\ell-1)} = \sigma_{\ell-1}\left(\sum_{k=1}^{n} w_{u,k}^{(\ell-1)} x_k + b_u^{(\ell-1)}\right) \tag{7}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input pattern. A generic target output for the DNN is given by

$$\hat{y} = \sigma_\ell\left(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} \hat{h}_u^{(\ell-1)} + b^{(\ell)}\right) \tag{8}$$

where $\hat{h}_u^{(\ell-1)}$ represents the target for u-th neuron in the hidden layer, for $u = 1, \ldots, d_{\ell-1}$. Since $z_u^{(\ell-1)} = \hat{h}_u^{(\ell-1)} - h_u^{(\ell-1)}$, we can write $\hat{h}_u^{(\ell-1)} = h_u^{(\ell-1)} + z_u^{(\ell-1)}$ and the latter, in turn, can be rewritten as

$$\hat{y} = \sigma_\ell\left(\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} (h_u^{(\ell-1)} + z_u^{(\ell-1)}) + b^{(\ell)}\right) \tag{9}$$

$$= \sigma_\ell\left(\underbrace{\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} h_u^{(\ell-1)}}_{\tilde{a}} + \underbrace{\sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}}_{\tilde{a}_z} + b^{(\ell)}\right) \tag{10}$$

where the quantities in the form $\tilde{a}_z$ are the activations (i.e., inputs) to the corresponding neurons, and will be defined shortly. Given the discussion so far, the target output can be written as

$$\hat{y} = y + y_z \tag{11}$$

where y_z is the output component related to the residues, namely, $y_z = \sigma_\ell(\tilde{a}_z)$; therefore

$$y_z = \hat{y} - y \tag{12}$$

$$\tilde{a}_z = \sigma_\ell^{-1}(y_z) \tag{13}$$

Starting from $\tilde{a}_z$, that is, $\tilde{a}_z = \sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}$, the residuals z_u for the lower layer(s) of the DNN can be computed as follows. First, when a generic pattern $\mathbf{x} \in \mathcal{D}$ is fed into the DNN, a certain output error y_z is observed. Different neurons in layer $\ell - 1$ may have diverse degrees of responsibilities for that particular error. Formally, the responsibility $r_u^{(i)}(\mathbf{x}) \in [0, 1]$ of the u-th neuron in the generic i-th layer having size d_i shall satisfy

$$\sum_{u=1}^{d_i} r_u^{(i)}(\mathbf{x}) = 1 \tag{14}$$

To this end, we define $r_u^{(i)}(\mathbf{x})$ as

$$r_u^{(i)}(\mathbf{x}) = \frac{\sigma_i(a_u)}{\sum_{k=1}^{d_i} \sigma_i(a_k)} \tag{15}$$

where a_u and a_k are the activations of the generic units u and k, respectively. In the following, for notational convenience, writing explicitly the dependence on $\mathbf{x}$ may be dropped whenever needed. Equation (15) allows for the computation of the residues $z_u^{(i)}$, $u = 1, \ldots, d_i$. It relies on the assumption that the higher the responsibility of a neuron on the overall error, the higher shall be the corresponding residue $z_u^{(i)}$ required to compensate for the misbehavior of the neuron at hand. Finally, we empirically factorize $\tilde{a}_z$ in terms of a sum of responsibility-weighted contributions from the neurons in the previous layer as $\tilde{a}_z = \sum_{u=1}^{d_{\ell-1}} w_u^{(\ell)} z_u^{(\ell-1)}$, where $r_u^{(\ell-1)} \tilde{a}_z = w_u^{(\ell)} z_u^{(\ell-1)}$, such that

$$z_u^{(\ell-1)} = \frac{r_u^{(\ell-1)} \tilde{a}_z}{w_u^{(\ell)}} \tag{16}$$

The pseudo-code of the overall procedure is presented in Algorithm 1, where a pattern-specific index j is used in order to make explicit the dependence of each quantity on the specific input vector.

Algorithm 1 Residual Driven Target Propagation (RDTP)

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

for $j = 1$ to N **do**
  $y_j \leftarrow F_{i+1}(\mathbf{x}_j)$
  $y_{z,j} \leftarrow \hat{y}_j - y_j$
  $\tilde{a}_{z,j} \leftarrow \sigma_{i+1}^{-1}(y_{z,j})$
  for $u = 1$ to d_i **do**
    $r_u^{(i)}(\mathbf{x}_j) = \frac{\sigma_i(a_u)}{\sum_{s=1}^{d} \sigma_i(a_s)}$
    $z_u^{(i)} = \frac{r_u^{(i)} \tilde{a}_z}{w_u^{(i+1)}}$
    $h_u^{(i)} = \sigma_i(a_u)$
    $\hat{h}_u^{(i)} = h_u^{(i)} + z_u^{(i)}$
  end for
  $\hat{\mathbf{h}}_{i,j} = (\hat{h}_1^{(i)}, ..., \hat{h}_{d_i}^{(i)})$
end for

2.1.2. Gradient-Based Target Propagation

By construction, RDTP can backpropagate targets to the (topmost) hidden layer only when the network has a single output unit. As a consequence, it is not suitable to further propagate targets from layer $\ell - 1$ to layer $\ell - 2$, unless layer $\ell - 1$ is one-dimensional (which is hardly the case). Therefore, RDTP can be applied only to traditional, single hidden layer MLPs having a single output unit. To overcome this limitation, an extended version of the algorithm is herein proposed, called gradient-based target propagation (GBTP). As in RDTP, the idea is to estimate the residual values $\mathbf{z}_{\ell-1}$ for layer $\ell - 1$ such that $\hat{\mathbf{h}}_{\ell-1} = \mathbf{h}_{\ell-1} + \mathbf{r}_{\ell-1} \odot \mathbf{z}_{\ell-1}$, where the residues are multiplied element-wise by the responsibility of the individual neurons. In the following, the residues are computed via gradient descent by letting

$$\hat{\mathbf{h}}'_\ell = \sigma_\ell\Big(W_\ell(\mathbf{h}_{\ell-1} + \mathbf{r}_{\ell-1} \odot \mathbf{z}_{\ell-1}) + \mathbf{b}_\ell\Big) \tag{17}$$

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

$$\mathbf{z}'_{\ell-1} = \mathbf{z}_{\ell-1} - \eta \frac{\partial \mathcal{L}(\hat{\mathbf{h}}_\ell, \hat{\mathbf{h}}'_\ell)}{\mathbf{z}_{\ell-1}} \tag{18}$$

that can be applied either in an online or batch fashion. Propagation of the targets to the preceding layers is straightforward, by iterating the procedure in a backward manner over the layer-specific parameters. Algorithm 2 presents the pseudo-code of GBTP, where the function $calculate\_layer\_resp(net, i, \mathbf{x})$ computes the responsibilities for i-th layer of the network as in Equation (15), in an element-wise fashion, in response to the input $\mathbf{x}$. The procedure $estimate\_residues(net, \mathbf{x}, \mathbf{r}_i, \hat{\mathbf{h}}_{i+1}, \mathbf{h}_i)$ computes the residues for i-th layer by iterating the application of Equation (18) until a stopping criterion is met. Note that the pseudo-code uses a pattern-specific index for representing all the quantities involved in the computation.

Algorithm 2 Gradient-based RDTP

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

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

2.2. The Algorithm for Growing and Training the DGNN

Building on the TP mechanisms, it is straightforward to devise the DGNN growing and learning procedure. First, the DGNN is generated as a feed-forward neural network (e.g., an MLP) with a single hidden layer. Hereafter, we write *h-size* to denote the number of hidden units. Such an initial shallow neural network is trained via BP over the supervised data sample $\mathcal{D}$. TP (either RDTP or GBTP) is then applied in order to estimate the target values for all the hidden neurons, and the corresponding values of the loss function is computed. Afterwards, the $\bar{k}$ hidden neurons having highest loss are selected (where $\bar{k}$ is a hyperparameter, see Section 3). These neurons and their input connections are replaced by a subnet *Sub* (this realizes the proper "growing step"). The subnet is built as a one-hidden-layer MLP having $\bar{k}$ output units (the i-th of which realizes the activation function $f_i(a_i)$ for the i-th neuron replaced by *Sub*), *h-size* hidden units, and as many input neurons as the dimensionality of the DGNN input layer. The subnet *Sub* is then trained via BP using the values yielded by TP as target outputs for the corresponding output neurons of *Sub*. The procedure is recursively applied: TP is used to estimate targets for the hidden neurons of *Sub*. For each neuron in the output layer of *Sub*, the corresponding loss is computed. The average loss values $\bar{\lambda}_h$ and $\bar{\lambda}_{Sub}$ are then determined by averaging over the losses yielded by the remaining original hidden units in the DGNN and by the output neurons of *Sub*, respectively. If $\bar{\lambda}_{Sub} \geq \bar{\lambda}_h$, then the $\bar{k}$ highest-loss neurons in the hidden layer of *Sub* are grown; otherwise, growing is applied to the $\bar{k}$ original hidden neurons having highest loss. In both cases, a new subnet is introduced (having the same architecture as *Sub*) and trained based on TP. The entire procedure is applied recursively to all the original hidden neurons that have not been grown yet, as well as to all the subnets already present in the DGNN. This recursive growing step is repeated until a stopping criterion is met (namely, an early sopping criterion based on the validation loss evaluated at the whole DGNN level). Finally, a global refinement [22] of the model may be carried out by means of an end-to-end BP-based retraining of the overall grown neural architecture over $\mathcal{D}$ (starting from the DGNN parameters learned during the growing process). The recursive growing procedure aims at developing architectures having a number of internal layers that suits the nature of the specific learning problem at hand. Instead of growing the architecture by simply adding new neurons to the single hidden layer, the DGNN expands individual neurons in a depth-wise manner. The highest-loss criterion for selecting the neurons to be replaced by subnets entails that only those portions of the architecture that are actually relevant to the learning task are eventually grown.

3. Results

Experiments were designed to (1) assess the effectiveness of the proposed approach, as well as to (2) demonstrate that the results achieved by the DGNN do compare favorably with (and, possibly improve over) the state-of-the-art techniques. Publicly available, popular datasets were used in the experiments. The datasets were drawn from the UCI repository [23]. They correspond to several application-specific classification problems,

spanning a range of different underlying characteristics (namely, the dimensionality of the feature space, the nature of the features involved, and the cardinality of the dataset). The specific datasets used, along with their characteristics and their bibliographic source, are summarized in Table 1. All of them consist of real-life data collected in the field, corresponding to specific real-world tasks. The Adult dataset [24] is a collection of records from the Census database describing professionals in terms of age, work-class, education, race, sex, etc. The task is predicting whether the income of a given professional exceeds \$50 K/annum or not. The Ozone dataset [25] consists of weather/climatic measurements (temperatures, wind speed, etc.) collected from 1998 to 2004 at the Houston, Galveston, and Brazoria area at 1–8 h intervals. The task is the detection of the ozone level. The Ionosphere dataset [26] is a collection of radar data collected by a phased array of 16 high-frequency antennas located in Goose Bay, Labrador. The task is the classification of the radar returns from the ionosphere as either positive (the radar returns show evidence of the presence of some type of structure in the ionosphere) or negative (returns do not show any such presence; the corresponding signals pass through the ionosphere). The Pima dataset [27] is a collection of diagnostic measurements (number of pregnancies the patient has had, BMI, insulin level, age, etc.) carried out at the US National Institute of Diabetes among female patients of Pima Indian heritage. The task is to diagnostically predict whether or not a patient has diabetes. The Wine dataset [28] is a sample of chemical analysis (such as the quantity of alcohol, of malic acid, of magnesium, etc.) over a number of wines grown in the same region in Italy but derived from different cultivars. The task is determining the origin of any given wine. Vertebral [29] is a biomedical dataset where orthopaedic patients are to be classified into three classes (normal, disk hernia, or spondylolisthesis) based on six biomechanical features. Finally, Blood [30] is a dataset extracted from the Blood Transfusion Service Center in the city of Hsin-Chu (Taiwan) that is used for the classification of a variety of different measurements characterizing different blood donations.

Table 1. Characteristics of the datasets used in the experiments.

	Adult	**Ozone**	**Ionosphere**	**Pima**	**Wine**	**Vertebral**	**Blood**
Cardinality	48,842	2536	351	768	178	700	748
Nb. of features	14	72	34	8	13	6	5
Reference	[24]	[25]	[26]	[27]	[28]	[29]	[30]

We adopted the same robust many-fold crossvalidation methodology (and the same partitioning of the UCI datasets into training, validation, and test sets, on a fold-by-fold basis) used in [31]. As in [31], the hyperparameters of each algorithm were selected using the validation fraction of each fold-specific subset. The hyperparameters were tuned via random search, selecting the specific configuration of hyperparameters that resulted in the minimum validation loss. Random search was applied to the selection of the DGNN hyperparameters as well. For all the training algorithms under consideration, the following stopping criterion was applied: stop training once the loss function evaluated on the fold-specific validation subset of the data has not shown a relative improvement of as much as (at least) 2% over the last 200 consecutive training epochs.

We first evaluated the improvement yielded by the DGNN over a standard DNN having the same initial architecture, namely, a three-layer DNN. Of course, the dimensionality of the input and output spaces are dataset-specific. The number of neurons per hidden layer was fixed according to the aforementioned model selection procedure applied to the plain DNN (the architecture was inherited by the DGNN as a starting point, before growing takes place), and it ranged between a minimum of eight neurons for the Adult dataset to a maximum of 30 neurons for the Ionosphere dataset. Table 2 compares the values of the average classification accuracies (and the corresponding standard deviations) yielded by the plain DNN with no growing mechanism (hereafter termed *base-model*) with the outcome of the *grown-model*, which is the same DNN whose architecture underwent growing during its training process. The accuracies are averaged over the test subsets of the many-fold

crossvalidation procedure for the different UCI datasets under consideration. The third and fourth columns of the Table report the average absolute accuracy improvement offered by the grown-model over the base-model and the corresponding relative error rate reduction, respectively. It is seen that the DGNN yields an improvement over the base-model for all the UCI datasets at hand. The improvement is significant: in fact, it amounts to an average 25.66% relative error rate reduction. The average improvement of the DGNN over the plain DNN in terms of absolute accuracy is significant as well, being approximately 3%. For five out of seven datasets, the DGNN turned out to also be more stable than the base-model, resulting in a reduced standard deviation of the fold-by-fold dataset-specific accuracies. In fact, on average (last row of the Table), the standard deviation of the grown-model is smaller than the standard deviation of the base-model. This is remarkable in light of the fact that the DGNN ends up becoming a more complex machine than the bare base-model, and its increased complexity (and architectural variance) could have been suspected of worsening the generalization capabilities of the resulting learning machine, affecting the stability of the latter. These results are empirical evidence of the fact that this is not the case. To the contrary, it is seen that the DGNN training algorithm tends to grow subnets that actually improve the quality of the mapping realized by the DGNN without overfitting the specific training data.

Table 2. Average accuracy ($\pm$std. dev.) on the test subsets of the many-fold crossvalidation procedure yielded by the base-model and by the grown-model, respectively, along with the average absolute accuracy improvement offered by the latter over the former and the corresponding relative error rate cut.

Dataset	Base-Model Accuracy (%)	Grown-Model Accuracy (%)	Avg. Absolute Gain (%)	Avg. Relative Error Cut (%)
Ionosphere	87.78 ± 2.03	93.47 ± 0.49	5.69	46.56
Wine	98.30 ± 0.98	99.43 ± 0.98	1.13	66.47
Vertebral	78.90 ± 2.96	87.01 ± 1.84	8.11	38.44
Blood	77.14 ± 0.58	80.35 ± 3.03	3.21	14.04
Pima	75.00 ± 2.68	77.08 ± 2.05	2.08	9.32
Ozone	97.24 ± 0.18	97.32 ± 0.27	0.08	2.90
Adult	85.38 ± 0.00	85.66 ± 0.00	0.28	1.92
Average	85.69 ± 1.34	88.62 ± 1.24	2.94	25.66

Table 3 reports the average absolute accuracy improvement (%) and the average relative error rate reduction (%) yielded by the proposed growing mechanism as functions of the dataset-specific number of features. It is seen that the improvements offered by the DGNN are substantially independent of the dimensionality of the feature space. In fact, Pearson's correlation coefficient r between the number of features and the average absolute accuracy improvement turns out to be $r = -0.3395$, which is a nonsignificant very small negative relationship between the two quantities (the p-value being equal to 0.4563). As for the correlation between number of features and average relative error rate cut, Pearson's coefficient is $r = -0.2317$ (p-value $= 0.6172$), i.e., an even less significant, very small negative correlation. In short, the performance of the DGNN growing algorithm is affected by the number of features to an extremely limited extent.

An illustrative instance of the sensitivity of the DGNN performance to the number $\bar{k}$ of neurons that are grown during the learning process can be observed graphically in the following figures. Figure 3 shows the average validation accuracy on the *Ozone* dataset yielded by the trained DGNN as a function of $\bar{k}$. Different curves are plotted in the figure, each corresponding to a different initial number of neurons in the DGNN hidden layer. The following remarks are in order: (1) as expected, regardless of the growing mechanisms, the accuracy of the DGNN is affected significantly by the initial size of the hidden layer; (2) the optimal (i.e., yielding maximum-accuracy) value of $\bar{k}$ strictly depends on the initial dimensionality of the hidden layer; (3) the accuracy of the resulting model is definitely not

a monotonic function of $\bar{k}$; neither does it present a unique maximum. Likewise, Figure 4 represents an overview of the variations of the validation accuracies on the remaining UCI datasets considered in the paper for different values of $\bar{k}$. A substantial variability in the behavior of the DGNN can be observed in the graphics, depending on the specific data at hand (and, implicitly, on the corresponding dimensionality), as expected.

The positioning of the learning and classification capabilities of the DGNN with respect to the state-of-the-art algorithms were assessed by means of two comparative experimental evaluations. The former aimed at putting the DGNN in the proper context of established results yielded by popular DNN-based approaches [32]. Hereafter, the established results are quoted from [32] (which obtained them via random-search model selection). Henceforth, the DGNN is compared with the following algorithm: self-normalizing neural networks (SNN) [32], sparse rectifier (s-ReLU) neural networks [33], deep residual neural networks (ResNet) [34], DNNs with batch-normalization (BN) [3], DNNs with weight normalization (WN) [35], and DNNs with layer normalization (LN) [36]. The outcomes of the experimental comparisons are reported in Table 4.

Table 3. Average absolute accuracy improvement and average relative error rate cut as functions of the number of features.

Dataset	Nb. of Features	Avg. Gain (%)	Avg. Error Cut (%)
Blood	5	3.21	14.04
Vertebral	6	8.11	38.44
Pima	8	2.08	9.32
Wine	13	1.13	66.47
Adult	14	0.28	1.92
Ionosphere	34	5.69	46.56
Ozone	72	0.08	2.90

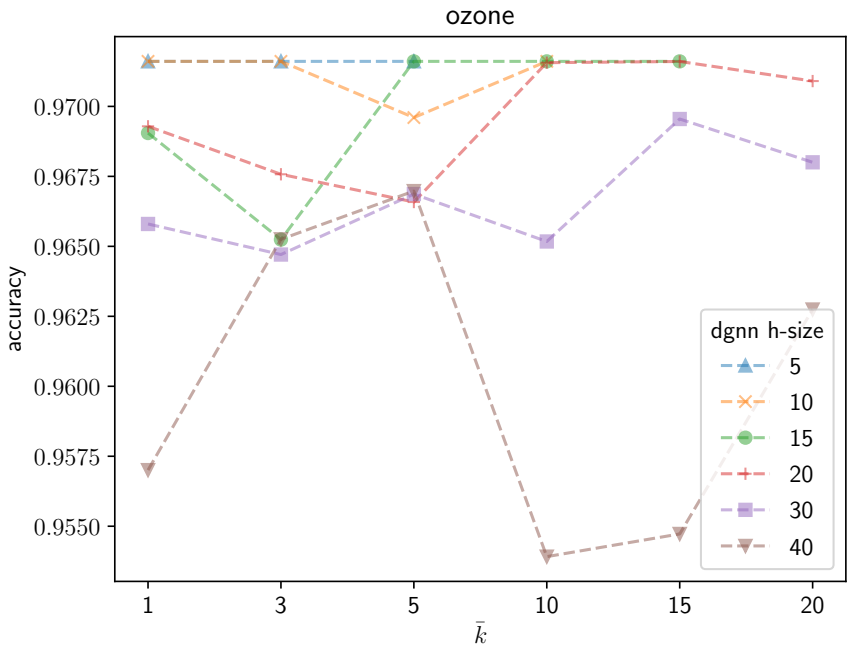

Figure 3. Accuracy yielded by the DGNN on the *Ozone* dataset as a function of the number $\bar{k}$ of neurons to be grown by the algorithm during the growing process, for different initial numbers of neurons (h-size) in the hidden layer.

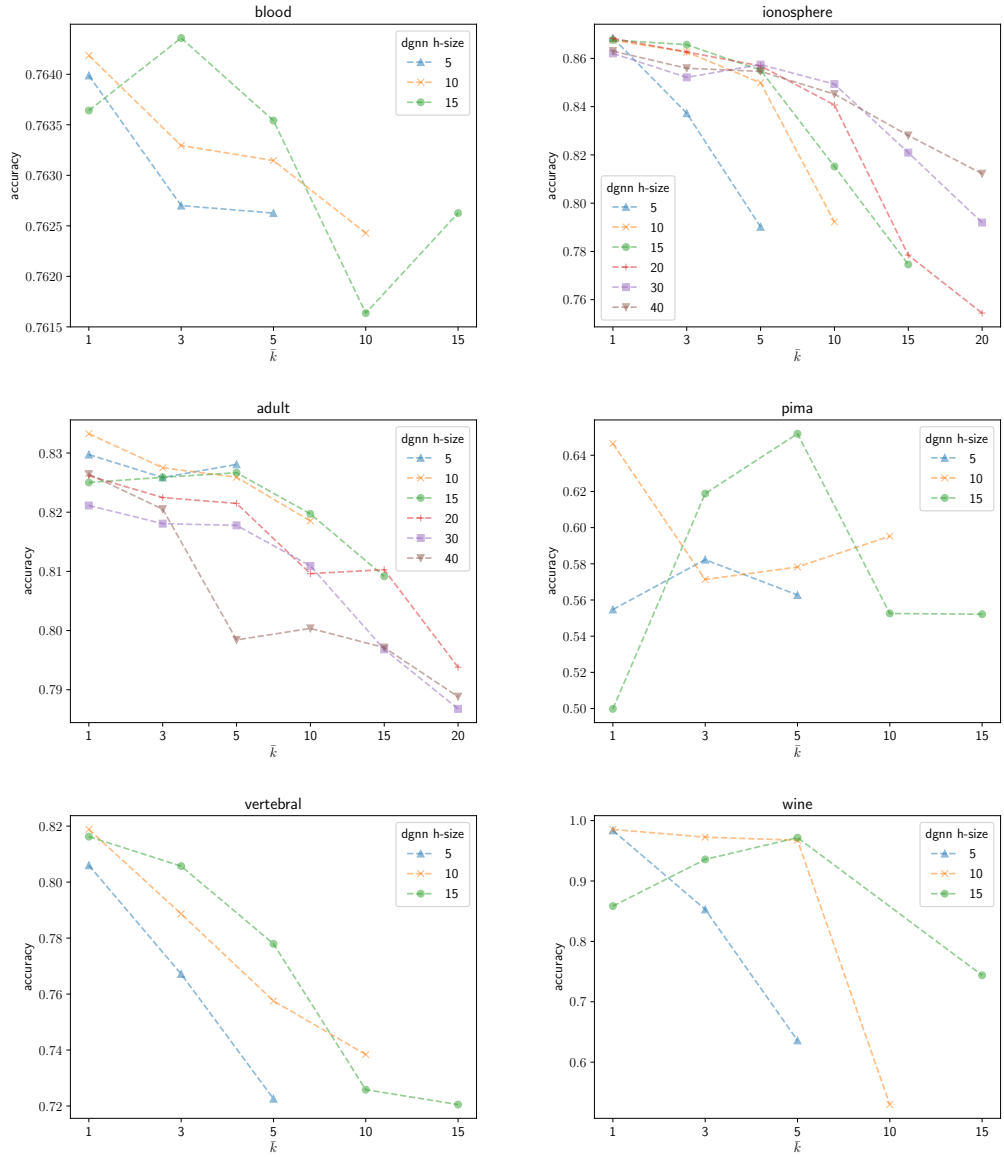

Figure 4. Accuracy of the DGNN on the UCI datasets as a function of $\bar{k}$.

Table 4. Comparison between the DGNN and the DNN-based classifiers: average accuracy on the different datasets. For each dataset, the highest accuracy is printed in bold.

Dataset	DGNN	SNN	s-ReLU	ResNet	BN	WN	LN
Ionosphere	93.47	88.64	90.91	**95.45**	94.32	93.18	94.32
Wine	**99.43**	97.73	93.18	97.73	97.73	97.73	97.73
Vertebral	**87.06**	83.12	87.01	83.12	83.12	66.23	84.42
Blood	**80.35**	77.01	77.54	80.21	76.47	75.94	71.12
Pima	**77.08**	75.52	76.56	71.35	71.88	69.79	69.79
Ozone	97.32	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
Overall average	**88.62**	86.25	86.77	87.06	86.46	83.55	85.67

The results are reported in terms of average accuracy over the many-fold crossvalidation procedure, on a dataset-by-dataset basis. For each algorithm, the last row of the Table presents the algorithm performance (i.e., the average accuracy) averaged over the different datasets. For each dataset at hand, a boldface font is used in the Table to highlight the models that resulted in the highest average accuracy. It is seen that the DGNN

resulted in the highest accuracy in five out of seven cases, and was second-best in the Ozone setup. In terms of overall average accuracy, the DGNN yielded a significant 1.56% gain over the ResNet, the latter being (on average) its closest competitor. Averaging over the six established DNNs reported in the Table, the DGNN resulted in an overall 2.66% average accuracy gain over its competitors. The two-tailed Welch's *t*-test resulted in a confidence >75% of the statistical significance of the gap between the accuracies achieved by the DGNN and those yielded by the ResNet. The confidence increases (>88%) when comparing the DGNN with the remaining approaches.

Table 5 reports the comparison between the DGNN and the other DNNs in terms of average depth (number of layers) and average number of hidden neurons. Note that the input and output neurons are not counted, insofar as they are implicitly defined by the nature of the specific datasets under consideration; hence, they do not affect the comparisons. The quantities in the Table are averaged over the different UCI datasets and over the different many-fold crossvalidation iterations. Except for the DGNN case, the other values are quoted from [32]. A lower-bound on the average number of layers needed for the ResNet was estimated based on the latter being reported in [32] as a 6.35-block network on average, where each such a block involved a minimum of two layers. Since the number of layers and the number of hidden neurons are indexes of the complexity of the neural networks at hand, it is seen that the average DGNN complexity is dramatically smaller than for all the other DNNs. This results in improved generalization capabilities, which is likely to be at the core of the accuracy gains yielded by the DGNN according to Table 4. In short, the DGNN growing mechanism tends to grow only those neurons whose growth actually benefits the learning and generalization capabilities of the DNN undergoing growing.

Table 5. Comparison between the DGNN and the DNN-based classifiers: average depth (number of layers) and average number of hidden neurons.

	DGNN	**SNN**	**s-ReLU**	**ResNet**	**BN**	**WN**	**LN**
Depth	4.04	10.80	7.10	>12.70	6.00	3.80	7.00
Nb. of hidden neurons	35	2765	1818	1626	1536	973	1792

In the second comparative experimental evaluation, we compared the DGNN with two established and popular growing algorithms for DNNs, namely, the adaptive structural learning of artificial neural networks (AdaNet) [12] and the cascade-correlation learning architecture (Casc-Corr) [7]. We adopted the same experimental setup used so far, namely, the same many-fold crossvalidation assessment procedure and the same model selection technique. The results are reported in Table 6. It turns out that the DGNN yields the highest average accuracies in four out of seven datasets, and the second-best accuracies in the remaining cases. When performing second-best, the difference between the accuracy yielded by the best scorer and the DGNN is negligible. In fact, upon averaging over the seven UCI datasets (last row of the Table) the DGNN yields the highest average accuracy overall. The two-tailed Welch's *t*-test shows that the statistical significance of the improvement yielded by the DGNN over its closest competitor (that is, the Casc-Corr) is quite high, with a confidence >95%.

Table 6. Comparison between the DGNN and the established growing algorithms: average accuracy on the different datasets. For each dataset, the highest accuracy is printed in bold.

Dataset	DGNN	AdaNet	Casc-Corr
Ionosphere	**93.47 ± 0.49**	77.56 ± 4.05	91.75 ± 2.82
Wine	99.43 ± 0.98	94.90 ± 1.88	**100.00 ± 0.00**
Vertebral	**87.06 ± 1.84**	71.44 ± 5.44	**87.06 ± 1.84**
Blood	80.35 ± 3.03	63.62 ± **1.89**	**80.90** ± 3.62
Pima	**77.08** ± 2.05	62.88 ± 3.90	76.80 ± **1.54**
Ozone	97.32 ± 0.27	**97.40** ± 0.20	97.10 ± **0.17**
Adult	**85.66 ± 0.00**	82.30 ± 0.26	84.75 ± 0.25
Average	**88.62 ± 1.24**	78.59 ± 2.52	88.34 ± 1.46

4. Conclusions

DGNNs extend plain DNNs insofar as they realize a growing mechanism that can adapt the architecture to the nature of the specific learning task at hand. Implicitly, such a growing mechanism results in the adaptation of the neuron-specific activation functions of the DNN, such that the activation function associated to a certain hidden neuron ξ is the (adaptive and nonlinear) function computed by the subnet associated with ξ. The best selling point of DGNNs over established growing algorithms lies in their locally expanding neurons only wherever necessary for improving the learning capabilities of the resulting machine, keeping the growth factor to a minimal scale which prevents the DNN from overfitting. From this perspective, DGNNs are complementary to the neural network pruning algorithms (which, by construction, start from an oversized architecture that is prone to overfitting since the early stages of the learning process).

The empirical evidence proves that the DGNN actually improves over established DNNs and growing algorithms, yielding sounder solutions to the different learning tasks covered in this paper. DGNNs offer practitioners a viable tool for compensating for possible mischoices at the architectural level made during the creation and initialization of the neural network. In principle, the DGNN growing mechanism may form the basis for overcoming any issues related to initial architectural choices.

An open issue with DGNNs is represented by the generation of target outputs for the different subnets. In fact, the target propagation process may not scale up to a very large dataset. Furthermore, the learning task entailed by the target propagation approach may also end up defining learning tasks for (at least some of) the subnets that are not necessarily simpler than the original learning task as a whole. Searching for solutions to these open problems is going to be part of the future work on the topic, alongside other research directions (e.g., the investigation of larger architectures for the subnets, the involvement of feature importance selection techniques for excerpting the neurons to be grown, etc.).

Finally, it is seen that the proposed approach introduces an implicit model of exploratory causal analysis, suitable to causal discovery [4,5] in the field. Let us resort to the probabilistic interpretation of artificial neural networks (the unfamiliar reader is referred to [37,38]). As shown in [39] (section 6.1.3, page 202), a feed-forward neural network realizes an implicit model of the conditional probability $p(\boldsymbol{t}|\boldsymbol{x})$ of the targets $\boldsymbol{t}$ given the input $\boldsymbol{x}$, where $\boldsymbol{t}$ and $\boldsymbol{x}$ are random vectors, insofar that $\boldsymbol{y}(\boldsymbol{x}) = \langle \boldsymbol{t}|\boldsymbol{x} \rangle = \int \boldsymbol{t} p(\boldsymbol{t}|\boldsymbol{x}) d\boldsymbol{t}$. As observed in [40] (section 5.6, page 272), the ANN may be used for solving either forward problems that "(...) correspond to causality in a physical system" (i.e., $\boldsymbol{y}(\boldsymbol{x})$ is caused by $\boldsymbol{x}$), or inverse problems (e.g., in pattern recognition it is the class $\boldsymbol{t}$ that causes the probability distribution of the features $\boldsymbol{x}$). Any generic hidden layer of the ANN realizes a nonlinear mapping $\varphi(.)$ of the random vector $\boldsymbol{x}$ onto a latent random vector $\boldsymbol{z} = \varphi(\boldsymbol{x})$. The specific observation $\boldsymbol{x}$ causes $\boldsymbol{z}$ and the latter, in turn, causes $\boldsymbol{y}$. Likewise, at each growing step the proposed algorithm introduces a new latent random vector that is caused by $\boldsymbol{x}$ and causes a selected subset of the components of the latent random vector yielded by the next hidden layer of the ANN, eventually causing $\boldsymbol{y}$, in a cascade-like fashion. In short, the DGNN discovers spontaneously new latent variables and a chain of causalities that better explain

the causality relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$. Moreover, since the algorithm selects subsets of the latent variables to be grown at each step, based on the extremization of the training criterion (that is, the best implicit fit of the resulting model to $p(\boldsymbol{t}|\boldsymbol{x})$), eventually different causal chains (of different length, involving different latent and observable variables) are discovered. At a post-processing stage, since each latent variable discovered by the DGNN is realized via a corresponding sub-network defined over $\boldsymbol{x}$, the connection weights in the sub-network that are below a given (small) threshold may be neglected, such that an overall pattern of causality between some input variables and specific latent and output variables emerges. Note that the forward-propagation of the input through such a grown DGNN complies with the two principles of Granger causality [41], insofar that (1) the feed-forward nature of the DGNN ensures that the causes happen prior to their effects, and (2) since the growing mechanism applies only to those variables that actually affect $\boldsymbol{y}$ and, in turn, the training criterion, the causes (along the causality chains) turn out to have unique information about the consequent values of their effects. It is seen that, due to its very architectural nature and dynamics, the DGNN and its generating data according to $p(\boldsymbol{t}|\boldsymbol{x})$ do relate to some extent to the notions of cellular automata and complex networks in the framework of causal calculus [42].

Author Contributions: V.L. and E.T. contributed equally to all facets of the present research. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Data Availability Statement: All data used in the paper are available publicly from the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/index.php (accessed on 31 January 2022).

Acknowledgments: The authors gratefully acknowledge the invaluable contributions from nobody. This paper is in memory of Ilaria Castelli, also known as Hillary Castles (LOL).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DNN	Deep neural network
AdaNet	Adaptive structural learning of artificial neural networks
BO	Bayesian optimization
BP	Backpropagation
BN	Deep neural network with batch-normalization
Casc-Corr	Cascade-correlation learning architecture
GBTP	Gradient-based target propagation
LN	Deep neural network with layer normalization
MLP	Multilayer perceptron
s-ReLU	Sparse rectifier linear unit
RDTP	Residual driven target propagation
ResNet	Deep residual neural networks
SNN	Self-normalizing neural networks
TP	Target propagation
WN	Deep neural network with weight normalization

References

1. Glorot, X.; Bengio, Y. 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, 13–15 May 2010; pp. 249–256.
2. Srivastava, N.; Hinton, G.E.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.

3. Ioffe, S.; Szegedy, C. 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; pp. 448–456.
4. Zenil, H.; Kiani, N.A.; Marabita, F.; Deng, Y.; Elias, S.; Schmidt, A.; Ball, G.; Tegnér, J. An Algorithmic Information Calculus for Causal Discovery and Reprogramming Systems. *iScience* **2019**, *19*, 1160–1172. [CrossRef] [PubMed]
5. Zenil, H.; Kiani, N.A.; Abrahão, F.S.; Tegnér, J.N. Algorithmic Information Dynamics. *Scholarpedia* **2020**, *15*, 53143. [CrossRef]
6. McCulloch, W.S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* **1943**, *5*, 115–133. [CrossRef]
7. Fahlman, S.E.; Lebiere, C. The cascade-correlation learning architecture. In Proceedings of the 2nd International Conference on Neural Information Processing Systems, Cambridge, MA, USA, 1 January 1989; Touretzky, D., Ed.; Morgan-Kaufmann: Burlington, MA, USA, 1990; Volume 2, pp. 524–532.
8. Fritzke, B. A growing neural gas network learns topologies. In Proceedings of the 7th International Conference on Neural Information Processing Systems, Cambridge, MA, USA, 1 January 1994; pp. 625–632.
9. Martinetz, T.; Schulten, K. A "Neural-Gas" Network Learns Topologies. *Artif. Neural Netw.* **1991**, *1*, 397–402.
10. Wang, G.; Xie, X.; Lai, J.; Zhuo, J. Deep growing learning. In Proceedings of the IEEE International Conference on Computer Vision, Venice, Italy, 22–29 October 2017; pp. 2812–2820.
11. Yoon, J.; Yang, E.; Lee, J.; Hwang, S.J. Lifelong Learning with Dynamically Expandable Networks. In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
12. Cortes, C.; Gonzalvo, X.; Kuznetsov, V.; Mohri, M.; Yang, S. 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; Proceedings of Machine Learning Research; Volume 70, pp. 874–883.
13. Wu, L.; Wang, D.; Liu, Q. Splitting Steepest Descent for Growing Neural Architectures. In Proceedings of the Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, 8–14 December 2019; Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R., Eds.; pp. 10655–10665.
14. Wu, L.; Liu, B.; Stone, P.; Liu, Q. Firefly Neural Architecture Descent: A General Approach for Growing Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, Virtual, 6–12 December 2020.
15. Evci, U.; van Merrienboer, B.; Unterthiner, T.; Pedregosa, F.; Vladymyrov, M. GradMax: Growing Neural Networks using Gradient Information. In Proceedings of the The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, 25–29 April 2022. Available online: https://OpenReview.net (accessed on 30 May 2022).
16. Laveglia, V. Neural Architecture Search by Growing Internal Computational Units. Ph.D. Thesis, Università degli Studi di Firenze, Firenze, Italy, 2019. Available online: https://hdl.handle.net/2158/1303131 (accessed on 10 March 2023).
17. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*; Rumelhart, D.E., McClelland, J.L., PDP Research Group, Eds.; MIT Press: Cambridge, MA, USA, 1986; Chapter Learning Internal Representations by Error Propagation; Volume 1, pp. 318–362.
18. Lee, D.; Zhang, S.; Fischer, A.; Bengio, Y. Difference Target Propagation. In Proceedings of the Machine Learning and Knowledge Discovery in Databases—European Conference, ECML PKDD 2015, Porto, Portugal, 7–11 September 2015; Proceedings, Part I; pp. 498–515. [CrossRef]
19. Castelli, I.; Trentin, E. Combination of supervised and unsupervised learning for training the activation functions of neural networks. *Pattern Recognit. Lett.* **2014**, *37*, 178–191. [CrossRef]
20. Castelli, I.; Trentin, E. Semi-unsupervised Weighted Maximum-Likelihood Estimation of Joint Densities for the Co-training of Adaptive Activation Functions. In Proceedings of the Partially Supervised Learning—Proceedings of the 1st IAPR TC3 Workshop, PSL 2011, Ulm, Germany, 15–16 September 2011; Revised Selected Papers; Springer: Berlin/Heidelberg, Germany, 2011; pp. 62–71. [CrossRef]
21. Castelli, I.; Trentin, E. Supervised and Unsupervised Co-training of Adaptive Activation Functions in Neural Nets. In Proceedings of the Partially Supervised Learning—First IAPR TC3 Workshop, PSL 2011, Ulm, Germany, 15–16 September 2011; Revised Selected Papers; Springer: Berlin/Heidelberg, Germany, 2011; pp. 52–61. [CrossRef]
22. Laveglia, V.; Trentin, E. A Refinement Algorithm for Deep Learning via Error-Driven Propagation of Target Outputs. In Proceedings of the Artificial Neural Networks in Pattern Recognition—Proceedings of tghe 8th IAPR TC3 Workshop, ANNPR 2018, Siena, Italy, 19–21 September 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 78–89.
23. Dheeru, D.; Karra Taniskidou, E. UCI Machine Learning Repository. 2017. Available online: https://archive.ics.uci.edu/ml/index.php (accessed on 31 January 2022).
24. Kohavi, R. Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, OR, USA, 2–4 August 1996; Simoudis, E., Han, J., Fayyad, U.M., Eds.; AAAI Press: Menlo Park, CA, USA, 1996; pp. 202–207.
25. Zhang, K.; Fan, W. Forecasting skewed biased stochastic ozone days: Analyses, solutions and beyond. *Knowl. Inf. Syst.* **2008**, *14*, 299–326. [CrossRef]
26. Sigillito, V.; Wing, S.; Hutton, L.; Baker, K. Classification of radar returns from the ionosphere using neural networks. *Johns Hopkins APL Tech. Dig.* **1989**, *10*, 262–266.

27. Smith, J.W.; Everhart, J.E.; Dickson, W.C.; Knowler, W.C.; Johannes, R.S. Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus. In *Proceedings of the Symposium on Computer Applications and Medical Care*; IEEE Computer Society Press: Los Alamitos, CA, USA, 1988; pp. 261–265.
28. Aeberhard, S.; Coomans, D.; de Vel, O. *Comparison of Classifiers in High Dimensional Settings*; Technical Report 92–02; Department of Computer Science and Department of Mathematics and Statistics, James Cook University of North Queensland: Douglas, Australia, 1992.
29. Berthonnaud, E.; Dimnet, J.; Roussouly, P.; Labelle, H. Analysis of the Sagittal Balance of the Spine and Pelvis Using Shape and Orientation Parameters. *J. Spinal Disord. Tech.* **2005**, *18*, 40–47. [CrossRef] [PubMed]
30. Yeh, I.C.; Yang, K.J.; Ting, T.M. Knowledge discovery on RFM model using Bernoulli sequence. *Expert Syst. Appl.* **2009**, *36*, 5866–5871. [CrossRef]
31. Fernández-Delgado, M.; Cernadas, E.; Barro, S.; Amorim, D. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.* **2014**, *15*, 3133–3181.
32. Klambauer, G.; Unterthiner, T.; Mayr, A.; Hochreiter, S. Self-normalizing neural networks. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; pp. 971–980.
33. Glorot, X.; Bordes, A.; Bengio, Y. Deep Sparse Rectifier Neural Networks. In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, FL, USA, 11–13 April 2011; pp. 315–323.
34. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
35. Salimans, T.; Kingma, D.P. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In Proceedings of the 30th International Conference on Neural Information Processing Systems, Red Hook, NY, USA, 5–10 December 2016; pp. 901–909.
36. Ba, J.L.; Kiros, J.R.; Hinton, G.E. Layer normalization. *arXiv* **2016**, arXiv:1607.06450.
37. Trentin, E.; Lusnig, L.; Cavalli, F. Parzen neural networks: Fundamentals, properties, and an application to forensic anthropology. *Neural Netw.* **2018**, *97*, 137–151. [CrossRef] [PubMed]
38. Trentin, E. Soft-Constrained Neural Networks for Nonparametric Density Estimation. *Neural Process. Lett.* **2018**, *48*, 915–932. [CrossRef]
39. Bishop, C. *Neural Networks for Pattern Recognition*; Oxford University Press: Cary, NC, USA, 1995.
40. Bishop, C.M. *Pattern Recognition and Machine Learning*; Springer: Berlin/Heidelberg, Germany, 2007.
41. Granger, C. Testing for causality: A personal viewpoint. *J. Econ. Dyn. Control* **1980**, *2*, 329–352. [CrossRef]
42. Zenil, H.; Kiani, N.A.; Zea, A.A.; Tegnér, J. Causal deconvolution by algorithmic generative models. *Nat. Mach. Intell.* **2019**, *1*, 58–66. [CrossRef]