Initialization and Plasticity of CEFYDRA: Cluster-first Explainable FuzzY-based Deep self-Reorganizing Algorithm

Javier Viaña[1][0000-0002-0563-784X], Stephan Ralescu[2][0000-0002-3969-1342],

Vladik Kreinovich[3][0000-0002-1244-1650], Anca Ralescu[4][0000-0002-7564-3540],

and Kelly Cohen[5][0000-0002-8655-1465]

[1,2,3,4,5] University of Cincinnati, Cincinnati OH 45219, USA
[1] Massachusetts Institute of Technology, Cambridge MA 02139, USA
vianajr@mail.uc.edu
vianajr@mit.edu

Abstract. The CEFYDRA is a network of units whose outputs are obtained using a fuzzy Takagi-Sugeno-Kang approach. At each unit, the information is clustered in fuzzy sets and then mapped using logistic functions and Cauchy membership functions. There are two primary contributions in this paper. The first is a set of suggestions for the initialization criteria of the parameters of a CEFYDRA. The second is a proposal for the self-reorganizing algorithm that modifies the location of the clusters of each unit as the algorithm is trained with gradient descent.

Keywords: Explainable AI, Neural Networks, Fuzzy Logic, Gradient Descent, Deep Learning.

1   Introduction

Currently, the initialization criteria of weights and biases of deep neural networks are arbitrary and naive [1-2]. For the most part, they are chosen stochastically from a distribution that depends on the number of connections that converge on each neuron. This means that during the first phases of training the predictions are not accurate. In fact, part of these preliminary iterations may not be useful, i.e., until there is a minimum logical organization of the parameters, there is no "effective" learning. Considering the impact that neural networks are having in today's world, it is convenient to define new initialization criteria that may improve the initial predictions and even reduce the computational cost of learning by suppressing unnecessary epochs. However, little can be done from the classic neural network concept based on weights and activation functions, which require this sort of initialization techniques.

A less explored but very attractive way to solve this problem is to use networks that leverage other mathematical strategies at the unit level. Of course, this not only implies a fundamental redefinition of the neural network concept, but also requires the development of the respective update formulas for its parameters. The CEFYDRA, a network derived from the algorithms proposed by [3-7], whose prediction system of

every unit is more complex than just adding the weights and applying an activation function, allows for this type of improvement in the initialization.

Another aspect that we study in this paper is the concept of self-reorganization or plastic modifications in the morphology of the algorithm's structure. This idea has proven to be excellent in multiple algorithms like self-organized feature maps [9], or adaptive control strategies [8]. Indeed, several researchers believe that Artificial General Intelligence (AGI) can only be achieved through the emergence of high-level reorganizations that result from the bottom-level interactions of a multiagent complex system. This concept of reorganization in algorithmic architectures has been led mainly by the field of bio-inspired evolutionary optimization systems [10-13]. In fact, the terrestrial natural mechanisms, that have been perfected over millions of years, are the best example of intelligent behaviors (and the only) that we know of. However, there is yet much that needs to be done in the field in order to realistically reproduce the level of intelligence that nature exhibits.

2   On How to Initialize the Parameters Without Prior Knowledge of the Hidden Features

The architecture of a CEYDRA is a network of units that leverage a fuzzy Takagi-Sugeno-Kang approach for inference [3-7]. The inputs of every unit are mapped by different multidimensional logistic functions and grouped in fuzzy clusters defined by Cauchy membership functions. All the parameters are optimized following a gradient descent learning.

In our formulation, the upper left index in parenthesis represents the layer reference (0 for the output layer, 1 for the first hidden starting from the right), and the bottom index represents the unit within that layer. We use $\mathbf{x_i}, \mathbf{y_i}, \hat{\mathbf{y}}_i$ and $\mathbf{t_i}$ to denote the ith input, output, predicted output, and the output of a generic unit, respectively. Finally, indexes k and j refer to the cluster and input dimension within the unit. The membership functions are obtained from Cauchy distributions,

$$\substack{(\lambda)\\h_\lambda}\mu_c(\mathbf{x_i}) = \frac{1}{1 + \left\| \frac{{}^{(\lambda+1)}\mathbf{t}(\mathbf{x_i}) - \substack{(\lambda)\\h_\lambda}\mathbf{a_c}}{\substack{(\lambda)\\h_\lambda}\mathbf{b_c}} \right\|^2} = \left[1 + \sum_{h=1}^{H} \left(\frac{{}^{(\lambda+1)}t_h(\mathbf{x_i}) - \substack{(\lambda)\\h_\lambda}a_{ch}}{\substack{(\lambda)\\h_\lambda}b_{ch}} \right)^2 \right]^{-1}. \tag{1}$$

In [4-5], the functions for the approximation of each cluster are linear, however, we want to constrain the output of each unit between 0 and 1. Thus, we use logistic functions instead, for a generic unit h_λ within layer λ,

$$\substack{(\lambda)\\h_\lambda}r_c(\mathbf{x_i}) = \frac{1}{1 + \exp\left(-\substack{(\lambda)\\h_\lambda}\mathbf{m_c} \cdot {}^{(\lambda+1)}\mathbf{t}(\mathbf{x_i}) - \substack{(\lambda)\\h_\lambda}n_c \right)}. \tag{2}$$

Therefore, the parameters of each cluster are $\substack{(\lambda)\\h_\lambda}\mathbf{a_c}, \substack{(\lambda)\\h_\lambda}\mathbf{b_c}, \substack{(\lambda)\\h_\lambda}\mathbf{m_c}$ and $\substack{(\lambda)\\h_\lambda}n_c$.

Unlike in the case of [4-5], which could be seen as a CEFYDRA of just an output layer, a complex CEFYDRA with hidden layers poses the following challenge: How should the parameters in the hidden layers be initialized? For those units in the output

layer we can use the method carried out described in [4-5], i.e., clustering the input-output space, and then fitting the logistic functions and the Cauchy membership functions. However, the units in the hidden layers have no prior information of what the hidden variables are or should be. Thus, they need to be initialized in a different way. Fig. 1 represents the difference in the initialization process of the parameters for the hidden layers and for the output layer.
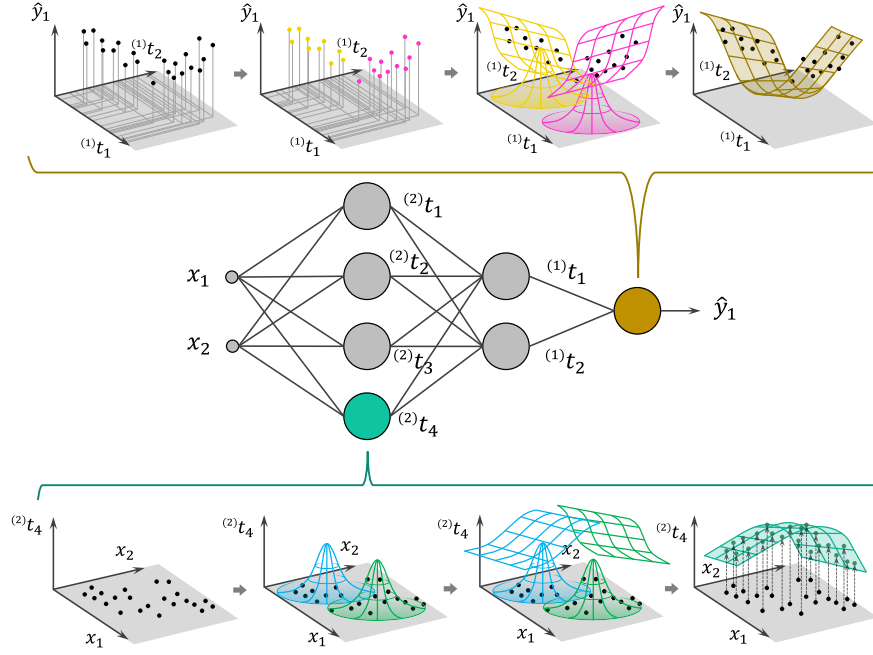

Fig. 1. Initialization steps for the parameters of a unit in the output layer (top flow) and the hidden layer (bottom flow). Example network of 4,3,1 configuration with 2 inputs. Initialization steps for the unit of the output layer: Cluster the joint input-output space and fit the logistic functions and the membership functions based on the clusters. Initialization steps for the unit of a hidden layer: Locate first the membership functions based on the input space and second the logistic functions, then map all the datapoints to proceed to the next layer.

This section focuses on the initialization method proposed for the parameters of both the Cauchy membership functions and the logistic functions of each unit in the hidden layers.

2.1 Initialization of the Cauchy membership functions.

We first find the centers of the Cauchy membership functions, which are represented by the parameter a_{kj}. Once all the centers of a particular unit have been chosen, we can then determine their amplitudes, defined by b_{kj}. It is necessary to incorporate a random component on the selection of the centers, otherwise, if it were done non-stochastically, all the units of a given layer could be identical, a situation that we want to avoid. The method we propose is to choose one of the training points randomly and

initialize the center of a given cluster so that it matches the location of such point. We would then repeat this operation until all the clusters of the unit have their centers located in the input space. It is also important to choose different points for each of the clusters to avoid having duplicated membership functions in the unit.

As it happens in every AI problem, it is convenient to get rid of any outliers in a pre-processing step. Otherwise, we could be initializing the center of the membership function in a location that is perhaps too far from the bulk of the data.

The next step is the determination of b_{kj}, we suggest studying each of the dimensions individually to assign a value to b_{kj}. Fig. 2 explains a technique to choose these values by considering the midpoints between each of the centers of the clusters. We use the average of the left and right separation (denoted by $s_{k,k+1}$), to obtain $b_{kj} = \frac{s_{k-1,k}+s_{k,k+1}}{2}$. This way, we can assure a fair distribution of the coverage of each membership function between the neighboring clusters without overlapping them more than the necessary. To those clusters that lie on the edges we directly assign the separation value of the only neighbor cluster they have, as seen in Fig. 2. Note that we can also calculate the widths of the middle clusters directly as $b_{kj} = \frac{a_{k+1}+a_{k-1}}{4}$.
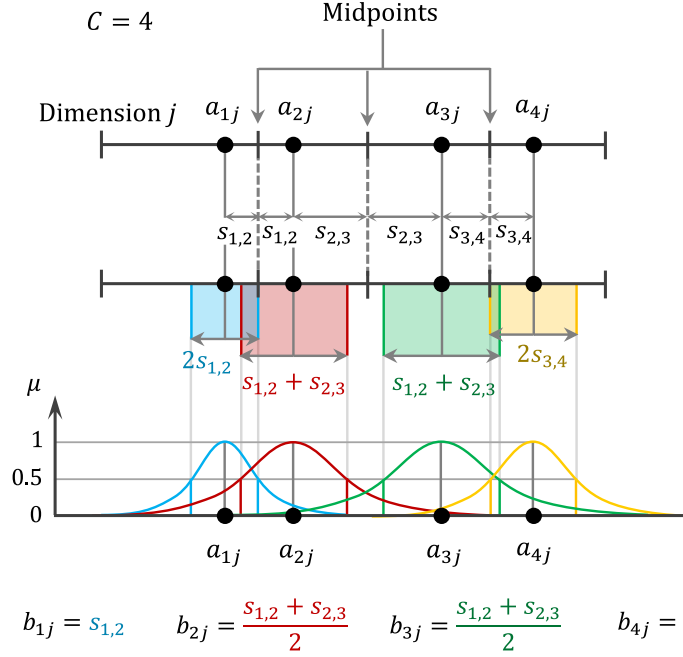

$$b_{1j} = s_{1,2} \qquad b_{2j} = \frac{s_{1,2}+s_{2,3}}{2} \qquad b_{3j} = \frac{s_{1,2}+s_{2,3}}{2} \qquad b_{4j} = s_{3,4}$$

Fig. 2. Calculation of the b_{kj} values based on the centers of all the clusters, a_{kj}, of a given unit.

The Algorithm 1 shows the pseudocode for these steps.

Algorithm 1: Initialization process for the Cauchy membership functions of a generic unit α in the hidden layer λ.

> **Inputs:** $^{(\lambda+1)}\mathbf{T}$ matrix of dimension $Q \times H_{\lambda+1}$, where Q is the number of $^{(\lambda+1)}\mathbf{t_i}$ training instances and $H_{\lambda+1}$ is the dimension of the unit's input space.
>
> **Outputs:** $^{(\lambda)}\mathbf{A}_\alpha$ matrix of dimension $C \times H_{\lambda+1}$, where C is the number of clusters, with the centers of the membership functions. $^{(\lambda)}\mathbf{B}_\alpha$ matrix of dimension $C \times H_{\lambda+1}$, with the amplitude of the Cauchy membership functions.

1 $^{(\lambda)}\mathbf{A}_\alpha, {}^{(\lambda)}\mathbf{B}_\alpha \leftarrow \mathbf{0}$
2 **for** each **cluster** k **in** C **do**
3   Choose a random point: $^{(\lambda)}\mathbf{A}_\alpha[k,:] \leftarrow {}^{(\lambda+1)}\mathbf{T}[\text{rand}(1,Q),:]$ **if not in** $^{(\lambda)}\mathbf{A}_\alpha$
4 **end**
5 **for** each **dimension** j **in** $H_{\lambda+1}$ **do**
6   Auxiliar vector of sorted centers: $\boldsymbol{d}_j \leftarrow \mathbf{sort}\ {}^{(\lambda)}\mathbf{A}_\alpha[:,j]$
7   **for** each **cluster** k **in range** 2 **to** C **do**
8          $$s_{k-1,j} \leftarrow \frac{\boldsymbol{d}_j[k] - \boldsymbol{d}_j[k-1]}{2}$$
9   **end**
10   First cluster: $^{(\lambda)}\mathbf{B}_\alpha[1,j] \leftarrow s_{1,2}$
11   Last cluster: $^{(\lambda)}\mathbf{B}_\alpha[C,j] \leftarrow s_{C-1,C}$
12   **for** each **cluster** k **in range** 2 **to** $C-1$ **do**
13         $$^{(\lambda)}\mathbf{B}_\alpha[k,j] \leftarrow \frac{s_{k-1,k} + s_{k,k+1}}{2}$$
14   **end**
15 **end**
16 **return** $^{(\lambda)}\mathbf{A}_\alpha, {}^{(\lambda)}\mathbf{B}_\alpha$

2.2   Initialization of the Logistic functions.

Now that we have the Cauchy membership functions initialized, we leverage them in order to initialize the logistic functions. We first divide the points of the training dataset (or a representative training sample to speed up the calculations) within each of the clusters that are now defined by each membership function. To do so, we evaluate every training datapoint and search for the maximum membership value, and then assign the point to the cluster whose membership value is the biggest, as shown in Fig. 3.
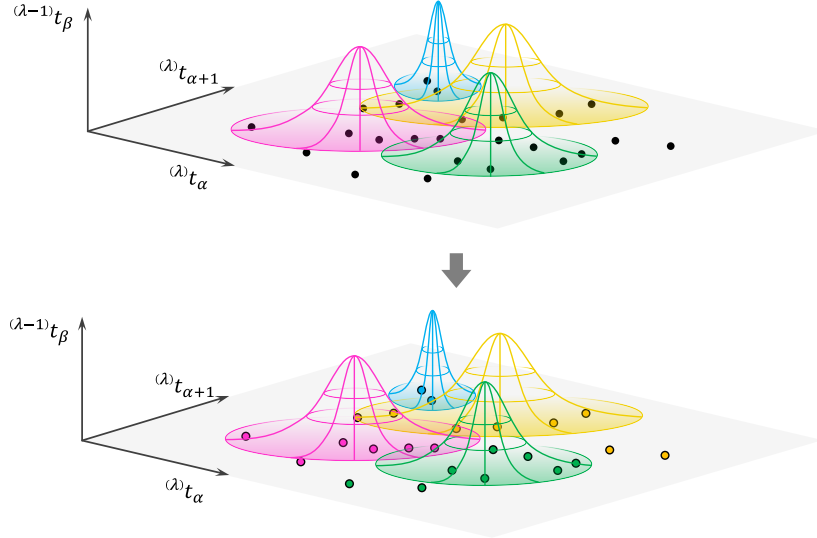
Fig. 3. Assigning of a cluster to the training datapoints based on the highest membership value of the Cauchy functions. Example of a two dimensional input space for visualization purposes, with 4 clusters.

Instead of searching for the parameters of a logistic function directly, we first choose the parameters of a linear function and then we approximate this function with the logistic function. This transformation allows us to perform an easier search of the parameters. Fig. 4 is a graphical demonstration of the accuracy in the approximation of different multidimensional linear functions with multidimensional logistic functions.
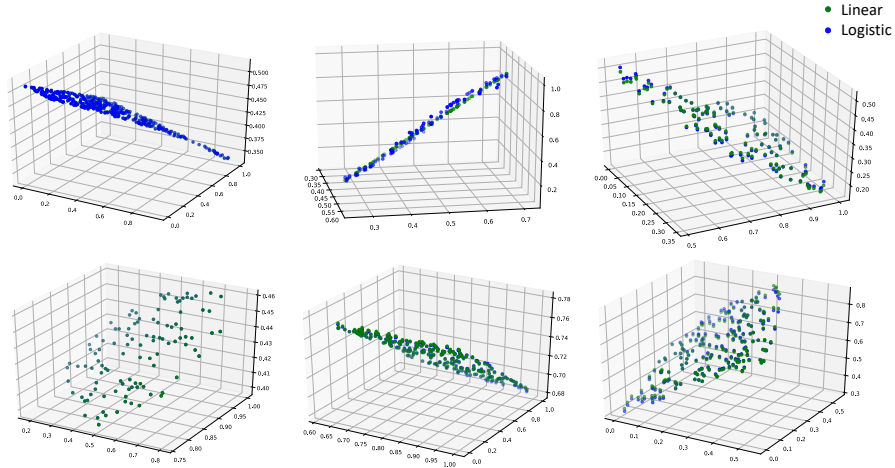

Fig. 4. Examples of approximation of the samples generated from six different linear functions (whose results have been normalized between 0 and 1) using multidimensional logistic functions.

Following with the initialization of the preliminary linear functions, for every dimension we get the minimum and maximum value of the points that belong to each cluster, in Fig. 5 identified as u and v, respectively.

$$u_{j,k} = \min\ {}^{(\lambda)}t_j\Big|_{\mu_k(^{(\lambda)}\mathbf{t}) \geq \mu_c(^{(\lambda)}\mathbf{t}), \forall c \neq k}$$

$$v_{j,k} = \max\ {}^{(\lambda)}t_j\Big|_{\mu_k(^{(\lambda)}\mathbf{t}) \geq \mu_c(^{(\lambda)}\mathbf{t}), \forall c \neq k}$$
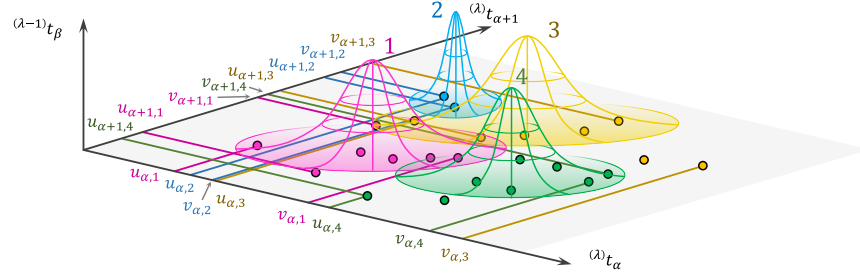

Fig. 5. Identification of the minimum and maximum values of the points in the clusters for every dimension. Example of 4 clusters, and 2 input dimensions, α and $\alpha + 1$, for visualization purposes.

Next, we introduce the concept of relative slope, which is the slope in the relative coordinates defined by every u and v pair, such that for a particular dimension, u is the relative origin and the distance $v - u$ is a unit of length.

For all the linear functions, we first choose randomly the values of the relative slopes, then we adapt the value of the relative slope to get the absolute slope using the u and v limits (Fig. 6), and finally we choose the value of the intercept. For the random selection of the relative slopes, for simplicity we decided to bound the random number between 1 and -1 (also to avoid dominance of a given slope over the others).
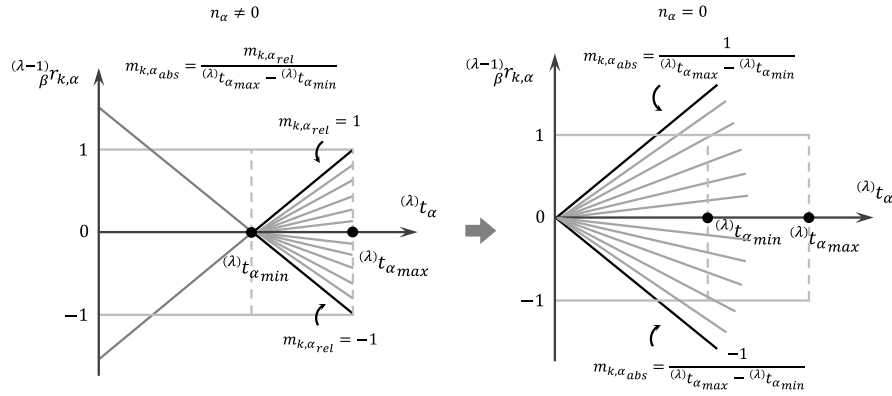

Fig. 6. Random choice of the relative slope in dimension α and the transformation to the absolute slope using the limits of the cluster ${}^{(\lambda)}t_{\alpha_{min}}$ and ${}^{(\lambda)}t_{\alpha_{max}}$.

Note that the use of a relative slope is essential to assure variance in the range of the data for dimensions where the difference ${}^{(\lambda)}t_{\alpha_{max}} - {}^{(\lambda)}t_{\alpha_{min}}$ is small. After all the absolute slopes of a linear function have been found, we still do not have a value for n. The intercept will be generated in the next step, where we normalize the entire hyperplane (as of now with a null intercept) between two values $\varkappa_{min}$ and $\varkappa_{max}$. These are both chosen randomly from a user-defined range, which we advise setting to [0,1]. First we need to map all the training points of the cluster to generate the output values, we do this using the current absolute value of the slopes. Then, we retrieve the minimum and maximum output values, ${}^{(\lambda-1)}_{\beta}r_{k_{min}} = \min_{{}^{(\lambda)}\mathbf{t_i}} \{\mathbf{m_{abs}} \cdot {}^{(\lambda)}\mathbf{t_i}\}$ and ${}^{(\lambda-1)}_{\beta}r_{k_{max}} = \max_{{}^{(\lambda)}\mathbf{t_i}} \{\mathbf{m_{abs}} \cdot {}^{(\lambda)}\mathbf{t_i}\}$ and we normalize the linear function so that those two limiting points are now $\varkappa_1$ and $\varkappa_2$, respectively. Thus, we obtain a new definition of the linear function, which we denote as ${}^{(\lambda-1)}_{\beta}r_k^{new}$, from the old ${}^{(\lambda-1)}_{\beta}r_k^{old}$,

$$\frac{{}^{(\lambda-1)}_{\beta}r_k^{new}(x) - \varkappa_1}{{}^{(\lambda-1)}_{\beta}r_k^{old}(x) - {}^{(\lambda-1)}_{\beta}r_{k_{min}}} = \frac{\varkappa_2 - \varkappa_1}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}}, \tag{3}$$

$$ {}^{(\lambda-1)}_{\beta}r_k^{new}(x) = \frac{(\varkappa_2 - \varkappa_1) \cdot {}^{(\lambda-1)}_{\beta}r_k^{old}(x)}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}} - \frac{(\varkappa_2 - \varkappa_1) \cdot {}^{(\lambda-1)}_{\beta}r_{k_{min}}}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}} + \varkappa_1, \tag{4}$$

$$ {}^{(\lambda-1)}_{\beta}r_k^{new}(x) = \frac{(\varkappa_2 - \varkappa_1) \cdot {}^{(\lambda-1)}_{\beta}r_k^{old}(x)}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}} + \frac{\varkappa_1 \cdot {}^{(\lambda-1)}_{\beta}r_{k_{max}} - \varkappa_2 \cdot {}^{(\lambda-1)}_{\beta}r_{k_{min}}}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}}. \tag{5}$$

Where $\frac{\varkappa_2 - \varkappa_1}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}}$ is the factor needed to multiply the absolute slopes to obtain the final values, and $\frac{\varkappa_1 \cdot {}^{(\lambda-1)}_{\beta}r_{k_{max}} - \varkappa_2 \cdot {}^{(\lambda-1)}_{\beta}r_{k_{min}}}{{}^{(\lambda-1)}_{\beta}r_{k_{max}} - {}^{(\lambda-1)}_{\beta}r_{k_{min}}}$ is the intercept.

Finally, we want to fit with logistic functions the sample instances obtained from the linear functions. To do so, we can transform the instances using a mapping of the type $T(x) = \log\left(\frac{1}{x} - 1\right)$ and then fitting them with a linear regression on the transformed space. The resulting slope vector and intercept obtained from this last fitting are the parameters of the final logistic expression. Note that x values of 0 or 1 should be discarded for the fitting, as they would lead to an error (one can instead apply an upper and lower threshold to avoid overflow issues).

Algorithm 2 shows the initialization process described for the logistic functions.

Algorithm 2: Initialization process for the logistic functions of a generic unit α in the hidden layer λ.

Inputs: $^{(\lambda+1)}\mathbf{T}$ matrix of dimension $Q \times H_{\lambda+1}$, where Q is the number of $^{(\lambda+1)}\mathbf{t_i}$ training instances and $H_{\lambda+1}$ is the dimension of the unit's input space.

Outputs: $^{(\lambda)}\mathbf{M_\alpha}$ matrix of dimension $C \times H_{\lambda+1}$, where C is the number of clusters, with the slope vectors. $^{(\lambda)}\mathbf{n_\alpha}$ vector with C entries, with the intercepts.

$^{(\lambda)}\mathbf{M_\alpha}, {}^{(\lambda)}\mathbf{n_\alpha} \leftarrow \mathbf{0}$

1   Variable to denote the points that belong to each cluster: $^{(\lambda)}\mathbf{p_\alpha} \leftarrow \mathbf{0}$

2   **for** each **point** $^{(\lambda+1)}\mathbf{t_i}$ in $^{(\lambda+1)}\mathbf{T}$ **do**

3   Get the dominant cluster for the point: $c \leftarrow \max_k \mu_k(^{(\lambda+1)}\mathbf{t_i})$

4   **to** $^{(\lambda)}\mathbf{p_\alpha}\{c\}$ **add** i

5   **end**

6   **for** each **cluster** k **do**

7   **for** each **dimension** j **in** $H_{\lambda+1}$ **do**

8   Get bottom limiting point: $^{(\lambda+1)}t_{j_{min}} \leftarrow \min_{^{(\lambda+1)}t_j(\mathbf{x_i})} {}^{(\lambda)}\mathbf{p_\alpha}\{k\}$

9   Get upper limiting point: $^{(\lambda+1)}t_{j_{max}} \leftarrow \max_{^{(\lambda+1)}t_j(\mathbf{x_i})} {}^{(\lambda)}\mathbf{p_\alpha}\{k\}$

10        $m_{rel} \leftarrow \text{rand}(-1,1)$

11        $\mathbf{m_{abs}}[j] \leftarrow \dfrac{m_{rel}}{^{(\lambda+1)}t_{j_{max}} - {}^{(\lambda+1)}t_{j_{min}}}$

12   **end**

13      $^{(\lambda)}_\alpha r_{k_{min}} \leftarrow \min_{^{(\lambda+1)}\mathbf{t_i}} \left\{ \mathbf{m_{abs}} \cdot {}^{(\lambda+1)}\mathbf{t_i} \right\}$

14      $^{(\lambda)}_\alpha r_{k_{max}} \leftarrow \max_{^{(\lambda+1)}\mathbf{t_i}} \left\{ \mathbf{m_{abs}} \cdot {}^{(\lambda+1)}\mathbf{t_i} \right\}$

15      $\varkappa_1, \varkappa_2 \leftarrow \text{rand}(0,1)$

16      $\mathbf{m'_{abs}} \leftarrow \dfrac{\varkappa_2 - \varkappa_1}{^{(\lambda)}_\alpha r_{k_{max}} - {}^{(\lambda)}_\alpha r_{k_{min}}} \cdot \mathbf{m_{abs}}$

17      $n'_{abs} \leftarrow \dfrac{\varkappa_1 \cdot {}^{(\lambda)}_\alpha r_{k_{max}} - \varkappa_2 \cdot {}^{(\lambda)}_\alpha r_{k_{min}}}{^{(\lambda)}_\alpha r_{k_{max}} - {}^{(\lambda)}_\alpha r_{k_{min}}}$

18   Map instances with linear function: $^{(\lambda)}\mathbf{p'_\alpha}\{k\} \leftarrow {}^{(\lambda)}\mathbf{p_\alpha}\{k\} \cdot \mathbf{m'_{abs}} + n'_{abs}$

19   Use the transformed space: $T(^{(\lambda)}\mathbf{p'_\alpha}\{k\}) \leftarrow \log\left(\dfrac{1}{^{(\lambda)}\mathbf{p'_\alpha}\{k\}} - 1 \right)$

20   Fit the transformed instances: $\mathbf{m_{final}}, n_{final} \leftarrow$ **fit linear** $T(^{(\lambda)}\mathbf{p'_\alpha}\{k\})$

21   Store the values of the slope vector and intercept:

22      $^{(\lambda)}\mathbf{M_\alpha}[k, :] \leftarrow \mathbf{m_{final}}$

23      $^{(\lambda)}\mathbf{n_\alpha}[k] \leftarrow n_{final}$

24   **end**

25   **return** $^{(\lambda)}\mathbf{M_\alpha}, {}^{(\lambda)}\mathbf{n_\alpha}$

26

In Fig. 7 and 8 we can see how the initialized mappings would look like for a CEFYDRA of 2 hidden layers and 2 units in each layer considering 4 and 10 clusters in each unit respectively.
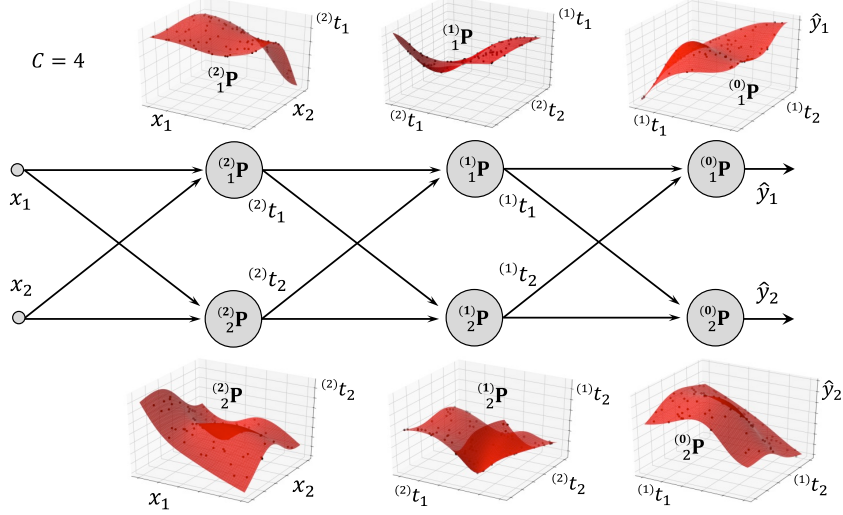

Fig. 7. Double input double output example to visualize the initialize mappings of each unit considering 4 clusters for each unit and 3 layers with 2 units per layer.
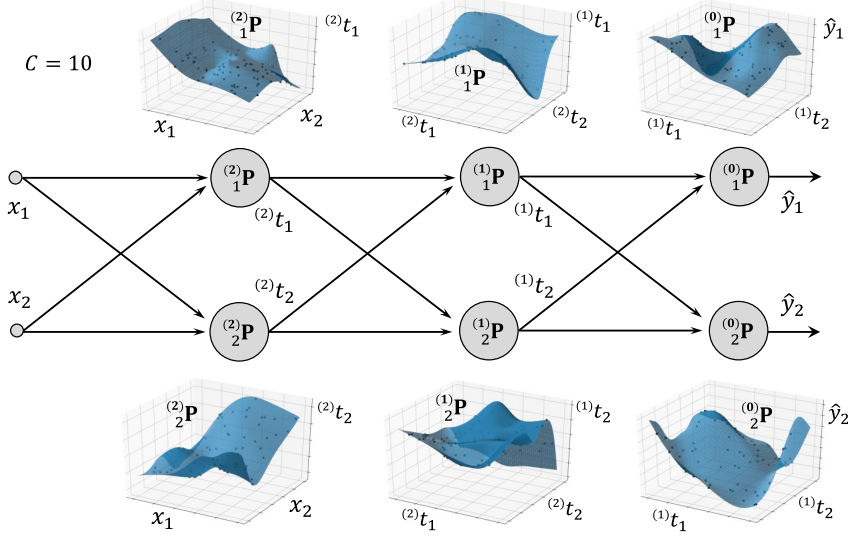

Fig. 8. Double input double output example to visualize the initialize mappings of each unit considering 10 clusters for each unit and 3 layers with 2 units per layer.

3   On the Plasticity of CEFYDRA

It should be noted that unlike a regular neural network where there are only weights and biases, in a CEFYDRA, there are four different types of parameters. The slopes and the intercepts of the logistic functions, and the centers and widths of the Cauchy membership functions. The last type, which we identify with b_{kj}, cannot be negative or null, because it defines the separation between the points of the function that reach the 0.5 membership value in the dimension j (let us remember that the Cauchy membership function is symmetric about its center of maximum membership value). If the CEFYDRA optimizes $\frac{1}{b_{kj}}$ and $-\frac{a_{kj}}{b_{kj}}$ instead of a_{kj} and b_{kj} (which would eliminate the denominator of the norm's argument in the Cauchy membership function), then none of the clusters would be able to reach a negative value of b_{kj} during the learning. Nonetheless, the width of a cluster can still become very small. In order to avoid any of these scenarios, a minimum threshold should be defined, $\diamond$, such that if it is crossed, we know that the cluster should be removed. Here resides the plasticity of the CEFYDRA; we delete any cluster that during the training satisfies $b_{kj} \leq \diamond$.

In order to keep constant the total number of clusters in the system, we also add a new cluster in a different unit of the network, thus avoiding any possibility of convergence to a naive system. This section aims to address the question of which unit should receive an additional cluster when a different cluster of the system has been deleted. To do so, we introduce the concept of clocks. Each clock is associated to a unit, and measures the cumulative change over the epochs that the unit has suffered. The clock of a unit is

$$\overset{(\lambda)}{_\alpha}\mathcal{H} = \sum_{e=1}^{E} \overset{(\lambda)}{_\alpha}\mathcal{H}e, \tag{6}$$

where E represents the total number of epochs till the moment, and

$$\overset{(\lambda)}{_\alpha}\mathcal{H}e = \frac{1}{C \cdot H_{\lambda+1}} \sum_{k=1}^{C} \sum_{j=1}^{H_{\lambda+1}} \left| \frac{\Delta \overset{(\lambda)}{_\alpha} w_{kj}}{\overset{(\lambda)}{_\alpha} w_{kj}} \right|. \tag{7}$$

Note that we use the absolute value of the relative change in the parameters (unlike $\overset{(\lambda)}{_\alpha}b_{kj}$, the parameters $\overset{(\lambda)}{_\alpha}a_{kj}$, $\overset{(\lambda)}{_\alpha}m_{kj}$, and $\overset{(\lambda)}{_\alpha}n_k$ are not necessarily positive).

Then, we use the method of the roulette wheel using the values of the clocks of all the units of the system to randomly assign a unit for the inclusion of a new cluster. Those clocks that have a greater value would have a greater probability of being selected, while those that have not changed much would have less probability. Fig. 9 shows an example of how these clocks might be at a given epoch for a CEFYDRA of 5 layers.
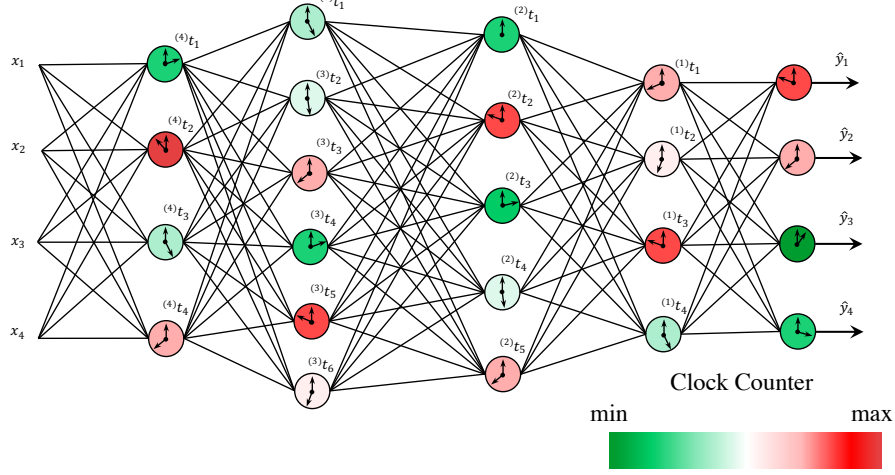
Fig. 9. Example scenario of the clocks for a system of 5 layers with 4, 6, 5, 4, and 4 units at a given epoch of the training phase.

Restarting the clocks every few epochs could be considered a good practice, since during the training process there might be phases when the units are evolving and others when they have converged. Thus, having a high clock value from the first epoch might not be representative of which unit are currently more subject to change. All in all, we make the assumption that the units which are changing the most are also the ones that deserve the highest chance to have a new cluster.

The clock of the unit that has deleted the cluster should be rebooted as well, in order to eliminate the chances of introducing its own cluster again, and also to provide the unit with a chance so it can evolve for the following epochs without that unnecessary cluster.

If at any point of the training process, a certain unit has only one cluster, then there would be no non-linearity in that unit and it wouldn't be contributing to the system. Thus, we could remove that cluster as well and assign it to a different unit. However, being conservative, there should also be some mechanism to limit the number of clusters of a certain unit to avoid converging towards the extremal case of having a single unit with all the initial clusters of the system. This would imply the generation of new units a certain locations, which we do not cover in this section but it would follow a similar principle to the one described for the inclusion of a new cluster.

When introducing a new cluster in a unit, we can follow the same procedure that we proposed for the initialization of the clusters' parameters: Random selection of a training point as the center of the cluster, using the neighbor clusters' centers to obtain the width of the membership function, and querying the points whose dominant cluster is the new one to initialize the slope and the intercept of the logistic function.

This concept of self-evolving morphology is certainly powerful in many applications that require adaptation of the system to better model the reality, rather than using a fixed definition of neurons and layers. Indeed, the entire structure of the algorithm changes autonomously with the learning, granting it the feature of plasticity.

The concept of explainability comes from the fact that we can retrieve the dominant clusters at each unit (since they are represented by the Cauchy membership function), and then combine all the dominant approximation functions to generate a final function that serves as an approximation of the inputs and outputs in the proximity of the instance. The precise algorithm to retrieve such analytic explanation will be covered in a different paper.

4   Conclusions

We proposed an initialization method for all the parameters of a CEFYDRA to guarantee enough diversity in the hidden outputs of the network during the first epochs of the training. We have seen that the mapping generated at every unit is significantly more complex than the mapping of a regular neural unit that leverages weights and an activation function. We also provided an algorithm for the self-reorganization strategy of CEFYDRA, which allows to relocate unnecessary clusters in other units of the network, and thus grants the system the ability to adapt its morphology to the data. Finally, we introduced the concept of clocks in order to measure the rate of change of every unit. These clocks serve as counters to decide for the most adequate relocation of the clusters. It could be argued that such technique resembles a certain natural evolutionary optimization algorithm. Future work should first focus on identifying the best example of a natural system that exhibits a similar self-evolving behavior, and then fine tune the proposed algorithm so it can be actually bio-inspired.

Acknowledgements

References

1. Springer J. M., Kenyon, G. T.: It's Hard for Neural Networks to Learn the Game of Life. In: 2021 International Joint Conference on Neural Networks (IJCNN), pp. 1-8 (2021).
2. Narkhede, M. V., Bartakke, P. P., Sutaone, M. S.: A review on weight initialization strategies for neural networks. Artificial Intelligence Review 55, 291–322 (2022).
3. Viaña, J., Ralescu, S., Cohen, K., Ralescu, A., Kreinovich, V: Localized Learning A Possible Alternative to Current Deep Learning Techniques. In: Melin, P., Castillo, O. (eds.) Studies in Computational Intelligence (2021).
4. Viaña, J., Cohen, K.: Fuzzy-Based, Noise-Resilient, Explainable Algorithm for Regression. In: Explainable AI and Other Applications of Fuzzy Techniques. 1st edn. Springer, Cham (2022).
5. Viaña, J., Ralescu, S., Cohen, K., Ralescu, A., Kreinovich, V.: Extension to multi-dimensional problems of a fuzzy-based explainable and noise-resilient algorithm". In: Proceedings of the 14th International Workshop on Constraint Programming and Decision Making CoProd'2021, Szeged, Hungary (2021).

14

6. Viaña, J., Ralescu, S., Cohen, K., Ralescu, A., Kreinovich, V.: Why Cauchy Membership Functions: Reliability. Advances in Artificial Intelligence and Machine Learning, (To Appear).

7. Viaña, J., Ralescu, S., Cohen, K., Ralescu, A., Kreinovich, V.: Why Cauchy Membership Functions: Efficiency. Advances in Artificial Intelligence and Machine Learning, 1(1), 81-88 (2021).

8. Kohonen, T.: Self-organization and associative memory, 8. Springer, Berlin (1989).

9. Verschure, F. M. J. P., Kröse, B. J. A., Pfeifer, R.: Distributed adaptive control: The self-organization of structured behavior. Robotics and Autonomous Systems 9(3), 181-196 (1992).

10. Arana-Daniel, N., Lopez-Franco, C., Alanis, A.: Bio-inspired Algorithms for Engineering. Elsevier, Amsterdam, Netherlands (2018).

11. S. Olariu, A.Y. Zomaya (Eds.), Handbook of Bioinspired Algorithms and Applications, Chapman & Hall/CRC (2006).

12. Simon, D.: Evolutionary optimization algorithms. John Wiley & Sons (2013).

13. Tonda, A. Inspyred: Bio-inspired algorithms in Python. Genet Program Evolvable Mach 21, 269–272 (2020).