A New Type of Non-Standard High Performance DNN with Remarkable Stability

Vincent Granville, *Co-Founder, BondingAI.io*
vincent@bondingai.io | linkedin.com/in/vincentg/
May 2025

***Abstract*—I explore deep neural networks (DNNs) starting from the foundations, introducing a new type of architecture, as much different from machine learning than it is from traditional AI. The original adaptive loss function introduced here for the first time, leads to spectacular performance improvements via a mechanism called equalization.**

To accurately approximate any response, rather than connecting neurons with linear combinations and activation between layers, I use non-linear functions without activation, reducing the number of parameters, leading to explainability, easier fine-tune, and faster training. The adaptive equalizer – a dynamical subsystem of its own – eliminates the linear part of the model, focusing on higher order interactions to accelerate convergence.

One example involves the Riemann zeta function. I exploit its well-known universality property to approximate any response. My system also handles singularities to deal with rare events or fraud detection. The loss function can be nowhere differentiable such as a Brownian motion. Many of the new discoveries are applicable to standard DNNs. Built from scratch, the Python code does not rely on any library other than Numpy. In particular, I do not use PyTorch, TensorFlow or Keras.

I. From standard to non-standard deep neural networks

A deep neural network is a system that represents a function $y = f(x, \theta)$ where x is the input, y is the response, and θ is the parameter. These components are usually multivariate. The notation $y = f_\theta(x)$ is also used, to emphasize the fact that f belongs to a parametric family of functions $\mathscr{F}_\theta$ indexed by θ. Given x and y, the goal is the find some θ^* in the parameter space $\mathscr{P}$ to minimize the distance $L(\theta) = ||f(x, \theta) - y||$. That is,

$$\theta^* = \underset{\theta \in \mathscr{P}}{\arg\min}\, L(\theta). \tag{1}$$

L is the loss function and $||\cdot||$ is the norm. The common norms L_1, L_2 lead respectively to the mean squared error (MSE) and mean absolute error (MAE). On occasions, I also use the maximum absolute error as an alternative to MAE.

Usually, the solution to (1) is not unique. This property is known as model non-identifiability [Wiki]. It makes it easier to find a solution; the drawback is lack of explainability in the solution and in the meaning of the parameters. I show later in this article how my original approach increases explainability.

One of the fundamental features of DNNs is the ability to choose $\mathscr{F}$ and $\mathscr{P}$ to approximate any solution – matching y to x – with arbitrary precision. In the context of standard DNNs, this is known as the universal approximation theorem (UTA) [Wiki]. It is achieved by using layered parameters with nested weighted sums. When moving one level down in the nested system, the weights being used are the subset of parameters attached to the next layer in the parameter set. To avoid ending up with a linear function f, a non-linear transform is applied to each weighted sum, at each level. This transform with values in $[0, 1]$ is called the activation function.

In the non-standard DNNs presented here, there is no intent to work with separable layers except in the parameter table. Weighted sums are replaced by non-linear transforms, eliminating the need for activation functions. And the universality theorem, even a stronger version of it, still applies.

A. Architecture of a non-standard DNN

Before diving into more advanced components of the architecture, let me introduce one of the simplest examples:

$$y = \sum_{k=1}^{m} \theta_{1,k} \Big[I_n - \theta_{2,k}^2 \Big(x - \theta_{3,k} I_n \Big)^2 \Big] \tag{2}$$

where x, y, I_n are column vectors with n components. Here n is the number of observations, and m is the number of features. In this example, we have 3 layers. The parameter θ is a $3 \times m$ matrix. More specifically:

- Layer l is $\theta_l = (\theta_{l,1}, \ldots \theta_{l,m})$ with $l \in \{1, 2, 3\}$.
- The vector I_n consists of ones exclusively.
- All operations are component-wise as with Numpy arrays. For instance, $(a, b)^2 = (a^2, b^2)$.
- All values in x and in the parameter table, are in $[0, 1]$.

To handle an input vector w with values outside $[0, 1]$, you can use the following mapping and inverse mapping:

$$w \mapsto x = \frac{w}{\sqrt{1 + w^2}}, \quad x \mapsto w = \frac{x}{\sqrt{1 - x^2}}. \tag{3}$$

Again, operations in (3) are done component-wise. In a simpler version, $\theta_{2,k}$ is the same for all k. Finally, it is easy to convert this system to a standard DNN.

Formula (2) approximates a Gaussian mixture model when all the elements are small (x, θ close to zero). In that case,

$$\exp\Big[-\theta_{2,k}^2 (x - \theta_{3,k})^2 \Big] \approx 1 - \theta_{2,k}^2 (x - \theta_{3,k})^2. \tag{4}$$

Approximation (4) leads to fast computations and also explains where formula (2) is coming from. In the Python code, a model similar to (2) is implemented not only when the input x is a vector, but also when it is an $n \times m$ matrix, in short, for typical tabular data. This applies to input data such as embeddings in large language models (LLM), see section II-D.

B. Gradient descent algorithm

The core engine of DNNs, including my system, is powered by a gradient descent algorithm. Iterations are called epochs. Starting with initial condition $\theta(0) \in \mathscr{P}$, the recursion is

$$\theta(i+1) = \theta(i) - \lambda_i \nabla L(\theta(i)). \tag{5}$$

The sequence of parameter estimates $(\theta(i))$ constitutes a non-chaotic discrete dynamical system. It is supposed to converge to a value θ^* solution to (1), thus minimizing the loss function L. Usually, the solution is not unique. Also, it could converge to a global minimum, or get stuck in a stable local minimum. The latter may be good enough for practical purposes. In the end, in case of true convergence (local or global), θ^* is a fixed point of the underlying dynamical system. Fake convergence or divergence – also known as vanishing gradient and exploding gradient – can happen due to numerical precision issues, or ill-conditioned computations. If L is a convex function, there is one global minimum and no local ones.

In my non-standard DNN, $\theta(i+1), \theta(i)$ and $\nabla(L(\theta(i))$ are $K \times m$ matrices. Here K is the number of layers and m is the number of features. Some layers may have fewer than m dimensions, resulting in non-rectangular matrices (with missing entries) with a variable number of columns. Formula (5) applies separately to each element in the parameter matrix. This format is efficient to deal with sparse matrices.

The symbol ∇ represents the gradient operator. When using the L_2 norm, it consists of partial derivatives. In the current version, the computation is as follows:

$$\frac{\partial L(\theta_\kappa(i))}{\partial \theta_\kappa(i)} = \frac{L(\theta_\kappa(i)+\epsilon) - L(\theta_\kappa(i)-\epsilon)}{2\epsilon} \tag{6}$$

where k is a bivariate index representing a location in the parameter matrix. With this generic formula, you don't need to know the explicit mathematical representation of L to compute its derivatives. In the current implementation, $\epsilon = 10^{-6}$.

With the L_1 norm, $\nabla(L(\theta(i))$ is replaced by νP where P is a random matrix of same shape as $\theta(i)$, consisting of uniform deviates in $[-2, 2]$, with ν being a simple normalization constant. I test q different versions of P (with a new set of random deviates each time) and choose the one that minimizes (5). If none of them results in $L(\theta(i+1)) < L(\theta(i))$ the proposed change if ignored. Thus, this is similar to rejection sampling. In practice, even $q = 1$ works. The number q may be larger depending on the dimension of P. Typically, L_1 runs faster than L_2. However, L_2 provides a better fit when it works well while L_1 works in almost all cases.

Finally, λ_i is the learning rate. For now, I use the same rate for all parameters at each iteration. Typically,

$$\lambda_i = 0.05 \quad \text{or} \quad \lambda_i = 0.10 \tag{7}$$

The learning rate may be decreased over time, using a decay mechanism governed by a hyperparamter called temperature. The higher the temperature, the more entropy in the system. In the code, the temperature is a function of MAE, which on average, decreases over time.

Before discussing other components, I conclude this section on the gradient descent algorithm with the following important points:

- Typically, each parameter must stay in some fixed interval, for instance $[0, 1]$, at all times. Thus, you need to use Lagrange multipliers in (6). The problem is known as constrained optimization and discussed in section 15.2 in [3].
- The initial value $\theta(0)$ consists of uniform deviates or a fixed value, say 0.5 for parameters in $[0, 1]$. Even a fixed value outside the boundaries may work in many cases. You may try different initial values and keep the one that provides the best fit. Always use a seed when generating random deviates, to guarantee replicability.
- At each iteration you can choose to climb rather than descend to avoid getting stuck too early in undesirable configurations. Either globally or for randomly selected axes. Make this choice with probability p. This is known as chaotic gradient descent and implemented in the code with $p = 0.30$.
- To predict the response $y(z)$ for z outside the input data x, use $y(z) = f(z, \theta^*)$. Note: θ^* depends on x and $y(x)$.
- Since most computations involve values in small intervals, say $\exp(w)$ with w in $[0, 1]$, you can use tabulated, pre-computed functions by small increments, for those called frequently and requiring more compute time. It dramatically speeds up computations and is similar to using quantization techniques. Easy to implement under L_1; with L_2 change ϵ from 10^{-6} to 10^{-2} assuming 10^4 pre-computed values per function.

There are several other ways to improve the gradient descent algorithm. See chapter 9 in [3] where I discuss an adaptive loss function that converges to the model evaluation metric, with very fast computations each time the loss $L(\theta)$ is updated. It produces plots like Figure 2, using a different loss function (closer to the evaluation metric) to boost the descent whenever it stalls.

To compute $\nabla L(\theta_i)$ in (5), when $L(\theta(i))$ is differentiable with respect to some parameters, but not to others, you can first use (6) for those where the partial derivative is properly defined (keeping the other ones fixed), and then the technique associated to L_1 for the parameters where the partial derivative does not exist, keeping the ones previously updated with (6) as fixed. I haven't tested this blended norm approach yet.

Gradient descent done on pure data, without knowledge of the underlying function nor its partial derivatives, is illustrated in section 15.4 in [3]. It works just as well: see Figure 1, and the corresponding video on YouTube, here, starting the descent from 100 different locations.

C. Reparameterization and latent features

The model characterized by (8) features a number of interesting concepts and properties, useful in specific applications:

$$y = \sum_{k=1}^{m} s\theta_{1,k} \left[s\theta_{2,k} I_n + \left(\frac{x - \theta_{4,k} I_n}{\theta_{3,k}} \right)^2 \right]^{-1} \tag{8}$$

As in model (2), the vectors x, y, I_n have n dimensions and I_n consists of ones only. Operations like square, multiplica-

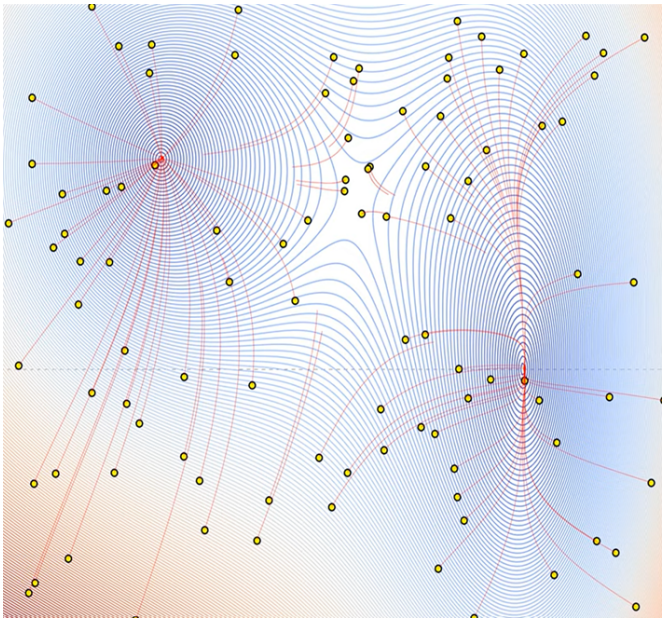

Fig. 1. Gradient descent on pure data; the mathematical formula for the curve is ignored

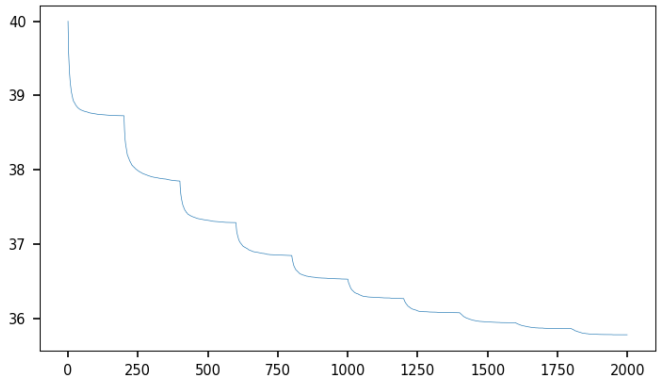

Fig. 2. Evolution of the loss (Y-axis) over 2000 epochs (X-axis), with adaptive loss function

tion, addition and division are performed component-wise as with Numpy arrays. All parameters and components of x are in $[0, 1]$ while s is a small, fixed positive constant. Clearly, we have 4 layers and the parameter matrix is full.

The origin of this model comes from kriging (spatial interpolation) or kernel density estimation using kernel functions [Wiki], here in n dimensions with m centers specified by the parameters $\theta_{4,k}$ with $k = 1, 2, \ldots, m$. The parameters $\theta_{3,k}$ are the kernel bandwidths. More specifically, adaptive bandwidths as they depend on the locations (centers). The parameters $\theta_{1,k}$ are weights, while the $\theta_{2,k}$ allow you to model each center as a singularity ($\theta_{2,k} = 0$) or smooth ($\theta_{2,k} = 1$) with something in-between when the value is not too close to zero. I tested various values of s, ranging from 10^{-4} to 1. Unlike classic DNNs, this non-standard DNN exemplifies explainable AI. It is also far more general than the machine learning models it is derived from. It also shares some similarity with non-Bayesian hierarchical models.

Looking at formula (8), the model is clearly overparameterized and non-identifiable, a feature it shares with almost all standard DNNs. It allows you to handle structures that would lead to instability in standard DNNs. The overparameterization makes the system more flexible, giving more chances (more pathways) for the gradient descent algorithm to find a decent local optimum. Yet, it does not lead to overfitting.

It is tempting to reduce the number of layers via reparameterization when possible, at least to speed up the computations. It rarely improves the fit. Sometimes, a different parameterization with same number of layers, produces better results. In this example, one obvious way to simplify (8) is as follows:

$$y = \sum_{k=1}^{m} s\theta'_{3,k} \left[sI_n + \left(\frac{x - \theta'_{1,k} I_n}{\theta'_{2,k}} \right)^2 \right]^{-1} \tag{9}$$

Now we have 3 instead of 4 layers. The mapping $\theta \mapsto \theta'$ is as follows:

$$\theta'_{1,k} = \theta'_{4,k}, \quad \theta'_{2,k} = \theta_{3,k}\sqrt{\theta_{2,k}}, \quad \theta'_{3,k} = \frac{\theta_{1,k}}{\theta_{2,k}}. \tag{10}$$

The mapping changes the distribution of the initial parameter values, from (say) independent and uniform on $[0, 1]$ to non-uniform and non-independent on some other domain. To keep independence and a uniform distribution on [0, 1] for the new parameters, an additional transform is needed. This transform depends on the mapping $\theta \mapsto \theta'$. You can use copulas to solve this problem, see chapter 5 in [4].

Finally, the unknown centers are latent parameters; the word hidden parameters is also used. They allow you to predict or reconstruct random spikes in semi-chaotic systems. There is an option in the code to choose fixed, prespecified centers instead. If the data is a one-dimensional time series, and the centers represent time locations with fixed increments, the system may represent a process with regular spikes of various amplitudes, well suited in applications such as fraud detection, extreme events, or cybersecurity. It does a good job detecting, modeling or predicting outliers with high accuracy. You may also use it for synthetic data generation in that context. See section II-C.

Note: In the context of computer vision, the same image filter can be represented either by a single-layer neural network with a large local window, say 40×40 pixels, or a very deep neural network with a tiny window (one adjacent pixel) and more than 500 layers. Both may produce the same results. See section 5.3 in [6]. Typically, one uses convolutional neural networks (CNNs) in this context.

D. Distillation, cross-validation, and noise injection

The applications discussed in section II could be described as model or curve fitting. To test and evaluate a model, you can use data points on a known curve in any dimension, then make it a realistic dataset via noise injection. This is one way to generate a rich collection of synthetic datasets. You can then test model sensitivity (impact on predictions) depending on the type and amount of noise, for various curves. This procedure is implemented in the code. Model evaluation is discussed in section III.

Besides reparameterization to reduce the number of layers, another compression technique is distillation. It consists of

removing at least 50% of the observations in the training data, or eliminating a large number of parameters. This option is available in the code. In practice, it has little impact on quality and sometimes even improve the results; yet it reduces computation time. An alternative is to reduce the number of features via feature clustering. This technique described in section 8.1 in [5].

Distillation is useful for cross-validation purposes. It allows you to assess the performance loss or gain (measured via MAE or MSE) when dropping 50% or more of your training data to make predictions on the full dataset.

E. Adaptive equalizer and other optimization techniques

The equalization technique described here transforms the fixed response y into an adaptive one. The response at epoch i is now denoted as $y(i)$. Thus, everything becomes moving parts, including the loss, now called an adaptive loss. After the last epoch, the final response is transformed back into its original: the same inverse transform applies both to the true response and the predicted one, whether estimated at x (the input data or training set), or at locations outside the training set. Starting with $y(0) = y$, it works as follows:

$$y(i+1) = \Phi_{\alpha_i}(y(i)) \tag{11}$$

where $\Phi_{\alpha_i} \in \mathscr{G}$, a parametric family of transforms satisfying $\Phi_\alpha^{-1} \in \mathscr{G}$ and $\Phi_\alpha \circ \Phi_\beta \in \mathscr{G}$ if both belong to $\mathscr{G}$. The simplest example besides rotations is the scale-location transform. In the current implementation, I simply use

$$y(i+1) = y(i) - \min(y(i)) \cdot I_n \tag{12}$$

where I_n is a vector consisting of ones, with the same shape as $y(i)$, that is, with n elements. Here the index i represents an epoch, not the location of an element in a vector.

In addition to accelerating the convergence under L_2, this mechanism takes care of the bias. It is easy to implement both in standard and non-standard DNNs. Also, it removes the first order terms in the approximation of y, allowing the DNN to do its job on second and higher order terms, explaining the accelerated convergence and fewer gradient problems. You can further improve by choosing a transform more sophisticated than (12). Also, you can use the same technique on x. I use the acronym E-DNN for equalized deep neural network, to name this technology.

Other optimization techniques include:

- stochastic gradient descent and batch processing, used in standard DNNs for the same purpose. Parameter estimates obtained from the latter can be combined from multiple batches at each iteration using ensemble methods. Batches also allow for parallel processing.
- The original chaotic gradient descent method explained in section I-B. It involves random ascents in some directions or sub-space of the parameter space, rather infrequently, to avoid getting stuck.
- Adaptive learning rates governed by a temperature hyperparameter decaying over time (see section I-B), with a different rate for each parameter layer.
- Feature augmentation. It consists of adding columns in the input data, either fake (synthetic) or related to the real features, to facilitate the job of the gradient descent algorithm. See section II-A.
- Proper handling of situations when the gradient descent produces a parameter value outside the accepted range at a given epoch. Reparameterization is another option.
- Testing different seeds or initial locations in the parameter space. Keeping the trained model obtained with the seed and/or initial configuration leading to the minimum loss. Likewise, stopping at the epoch associated with minimum loss, rather than after a pre-specified number of epochs.

Finally, in the gradient descent, at each epoch, you can update a subset of layers while keeping the parameters in the other layers unchanged. At the next epoch, you update another subset of layers keeping those processed in the previous step, unchanged. And so on, covering all layers in turn in a small number of successive epochs. You repeat this process iteratively. One way to manage it is as follows:

- Each epoch is broken down into sub-epochs (an inner loop within an epoch). A sub-epoch applies the gradient descent to a specific subset of layers, that is, to the parameters attached to these layers. Without even computing the partial derivatives associated to the layers outside this subset.
- A full epoch cycle is completed once all layers have been processed. How layers are grouped and the order in which they are processed within an epoch, can have a significant impact on the convergence.

The EM algorithm proceeds similarly. However, in my system, you can have more than 2 steps (sub-epoch) per iteration. Sub-epochs are not implemented yet. But it is easy to adapt my code to allow for it: in the gradient descent, specify which layers to work on at any given time. This eliminates the need for an inner loop.

II. Case Studies

Standard and non-standard DNN are equivalent. Let's illustrate with model (8). You start with the transformed input data x that takes values (say) in [0, 1]. Then the layer sequence is as follows:

$$\begin{aligned}
x &\mapsto x[1,k] = x - \theta_{4,k} I_n && k = 1, \ldots, m \\
&\mapsto x[2,k] = \theta_{3,k}^{-1}\, x[1,k] && k = 1, \ldots, m \\
&\mapsto x[3,k] = s\theta_{2,k} I_n + (x[2,k])^2 && k = 1, \ldots, m \\
&\mapsto x[4,k] = s\theta_{1,k} (x[3,k])^{-1} && k = 1, \ldots, m \\
&\mapsto y = \sum_{k=1}^{m} x[4,k]
\end{aligned}$$

The above notation is simplified to facilitate understanding. In fact, each $x[l,k]$ is an n-dim vector or a matrix if x is a matrix. So, I am implicitly using tensors, without reliance on TensorFlow or similar libraries. Also, there is no apparent activation function. These are used to break linearity when all

other transforms are linear. In my system, since most transforms are non-linear, you may say that activation functions are hidden, unknown, and irrelevant, yet dominate the architecture. In the end, standard and non-standard DNNs can approximate the same functions – any function – with arbitrary precision, yet using different representations.

Another main difference is the absence of backpropagation or forward-propagation mechanism in non-standard DNNs as the system is processed whole rather than in a layered fashion. It reduces the risks of gradient vanishing, getting stuck, and error propagation that arises in DNNs with many layers.

Finally, I keep parameters within their range, typically $[0, 1]$, as much as possible, even when the L_2 descent forces you outside. See how I do it in the code, in the last four lines of the `partial_derivatives` function. It significantly contributes to the robustness of the architecture.

Now that the connection to classic deep neural networks is established, I focus on the case studies.

A. *The workhorse of non-standard DNNs*

I call it the workhorse because it is a simple yet versatile model that works well with little if any fine-tuning, converges in fewer epochs than others discussed in this section, easily handles a large number of features, and covers many types of smooth responses. Also, I implemented it for tabular data: the input x can be a vector or a matrix. It is characterized by

$$y = \sum_{l=1}^{q}\sum_{k=1}^{m} \theta_{3l,k}\Big[I_n - \theta_{3l+1,k}^2\Big(x_k - \theta_{3l+2,k}I_n\Big)^2\Big] \qquad (13)$$

where x_k is the k-th feature, that is, a vector with n observations in the input data x. It generalizes (2) to tabular data and likewise, its root is in representing any multivariate function by a mixture of Gaussians, with the exponential replaced by its first two terms in the Taylor expansion, when x is close to zero post-transform.

The parameter matrix P has $3q$ rows and m columns. The model is encoded as q layers, each with 3 sub-layers. Each sub-layer has a full set of m parameters. The following sub-matrix represents the q centers (one per row):

$$P_{2:3} = \begin{bmatrix} \theta_{5,1} & \theta_{5,1} & \dots & \theta_{5,m} \\ \theta_{8,1} & \theta_{8,1} & \dots & \theta_{8,m} \\ \vdots & \vdots & & \vdots \\ \theta_{3l+2,1} & \theta_{3l+2,1} & \dots & \theta_{3l+2,m} \\ \vdots & \vdots & & \vdots \\ \theta_{3q+2,1} & \theta_{3q+2,1} & \dots & \theta_{3q+2,m} \end{bmatrix}$$

A better, more compact representation for P is to use a 3-D matrix. That is, a tensor. This may lead to faster computations.

Model (13) underperforms when the number of features and centers is very small, or when the number of centers is much larger than the number of features. The latter issue can be addressed by reducing the number of centers, or adding artificial features to the input data. Adding fake features consisting of noise, is called feature augmentation. While untested, using a copy of a real feature as a fake feature, may help. But in the end, it reduces explainability.

Model (13) is an approximation of some reality. However it requires strict constraints on the parameters to work well. The "exact" version (reparameterized) is recommended for most applications. It is characterized by

$$y = \sum_{l=1}^{q}\sum_{k=1}^{m} \theta_{3l,k} \exp\Big[-\Big(\frac{x_k - \theta_{3l+2,k}I_n}{\theta_{3l+1,k}^2}\Big)^2\Big] \qquad (14)$$

In (14), the parameters $\theta_{3l+2,k}$ are in the denominator instead yet still in $[0, 1]$. You can introduce a 4-th sub-layer as follows:

$$y = \sum_{l=1}^{q}\sum_{k=1}^{m} \theta_{4l,k} \exp\Big[-\theta_{4l+3,k}^2\Big(\frac{x_k - \theta_{4l+2,k}I_n}{\theta_{4l+1,k}^2}\Big)^2\Big] \qquad (15)$$

With all parameters in $[0, 1]$, it allows you to represent any function, without covariance matrix in the kernel. In short, model (13) uses an Epanechnikov kernel, while (14) is based on a Gaussian kernel. The former is optimal relative to MSE. Alternatively, replace $\theta_{4l+3,k}$ by $1 - \theta_{4l+1,k}$ in (15), reducing the number of sub-layers to 3 yet keeping the model as generic.

Models (13) and (14) are represented by the function `f0` in the code, with `model` set to `'approx. gaussian'` for the former, and `'gaussian'` for the latter. Figures 3, 4 and 5 show examples of response y that it can generate. The contour maps feature the first two coordinates in the input data set x. A small amount of noise is present in each case.

B. *A new type of universal approximation*

Besides standard DNNs and non-standard model (13), there are several ways to approximate an arbitrary data-driven response y which is not too chaotic. This is also true for chaotic systems – the topic of another discussion: see [2] and [7]. In one dimension, the general framework for additive models is

$$y = \sum_{k=1}^{m} \psi_k(x, \theta) \qquad (16)$$

where y is the response and x is the input data. Both are vectors with n rows (the observations) while θ is a multivariate parameter.

Formula (16) generalizes to multidimensional data, where x and sometimes y are matrices, or even when there is no y (see first chapter in [6]). Even to cases when there is no mathematical function behind the scenes, such as in exact data-driven multivariate interpolation: see chapter 4 in [5].

Here I focus on the 1-D case. When $\psi_k(x, \theta) = \theta_k x^k$, we are dealing with Taylor series and polynomial regression; θ_k is the k-th component of θ. This is possibly the worst example, whether $|x| < 1$ or not. More robust techniques include Fourier regression and regression with orthogonal polynomials, both discussed in chapter 4 in [5].

But what if we could find a function f that approximates any smooth response y as accurately as you want, with just one univariate parameter? A real number θ, not a vector, nor a matrix or a tensor. Actually, there are plenty of such universal functions. The most notorious one is

$$y \approx f(x, \theta) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{1}{k^x} \cos(\theta \log k) \qquad (17)$$

With few restrictions, it approximates any smooth y as accurately as you want, depending on θ. It requires $x \in]\frac{1}{2}, 1[$. This result is a consequence of the universality of the Zeta function [Wiki]. It is linked to the Riemann Hypothesis (RH). However, as discussed in chapter 17 in [6], the convergence of the series (18) is very slow and chaotic when θ is large and $x < 1$, and you need extremely large θ to get good approximations. This result, just like the the universal approximation theorem for DNNs, is not practical.

Instead, I turned (18) into a great tool by swapping the roles of x and θ, and introducing a large number of parameters, instead of a single real number. It leads to the following model:

$$y = \sum_{k=1}^{m} (-1)^{k+1} \frac{\theta_k}{k^\sigma} \cos\Big[\rho(x+\tau)\log k\Big], \tag{18}$$

$$y' = \sum_{k=1}^{m} (-1)^{k+1} \frac{\theta_k}{k^\sigma} \sin\Big[\rho(x+\tau)\log k\Big]. \tag{19}$$

Its parameter table has $m+3$ elements, and 4 layers separated by a semicolon in (20):

$$\theta = (\theta_1, \ldots, \theta_m;\ \sigma;\ \tau;\ \rho). \tag{20}$$

All the parameters are in $[0, 1]$. A more general version has specific $\sigma_k, \rho_k, \theta_k$ in the k-th term, rather than constants, thus leading to a full rather than incomplete parameter matrix. Also, you can use another θ (say θ') for y'. However, this is usually not needed and the incomplete model is more stable. Indeed, in my example, I even set $\rho = 1$, reducing the number of layers to 3.

Unlike Fourier regression, the model is both non-periodic and truly non-linear. It is also more generic. In most applications, you can ignore y' but here I need it for a specific purposes: modeling 2-D orbits. I estimate the parameters on y and then use the same values for y'. Note that when m is infinite, (18) and (19) are respectively the real and imaginary parts of the Dirichlet eta function in the complex plane. In the code, they are implemented as the `f2` and `f3` functions. The parameter τ is represented as $\varphi/(1-\varphi)$ with φ in $[0, 1]$ to allow for arbitrary large values.

C. Dealing with singularities and extreme events

Here I introduce a 4-layer model covering a large spectrum of responses, ranging from smooth to chaotic. The input data is 1-D. The formula is as follows, with $s = 10^{-3}$:

$$y = \sum_{k=1}^{m} s\theta_{1,k} I_n \Big[s\theta_{2,k} I_n + \Big(\frac{x - \theta_{4,k} I_n}{\theta_{3,k}}\Big)^2\Big]^{-1} \tag{21}$$

This model is overparameterized. For a 3-layer version, see (9) and (10). Even though the 3-layer is just as generic, it does not perform as well. In this case, parameter redundancy (that is, more layers) is a benefit. Despite having partial derivatives, the L_2 approach underperforms compared to my L_1 solution. The latter relies on swarm optimization for the gradient descent. As in example (14), this explainable AI model is easy to interpret and fine-tune with the intuitive parameters:

- **Weight** parameters: $\theta_{1,k}$. They can be positive, negative, or zero. In the code, increasing m but setting the additional weights to zero artificially inflates the number of training parameters, but can increase performance.
- **Offset** parameters: $\theta_{2,k}$. The closer to zero, the stronger the spikes or the dips. Figure 13 is produced with values close to zero.
- **Center** parameters: $\theta_{4,k}$. Assumed as unknown (latent) in the default version, with `model='latent'`. Easier to train with fixed centers, that is, with `model='static'` in the code.
- **Skewness** parameters: $\theta_{3,k}$. It has the same meaning as the standard deviation σ in a Gaussian kernel.

In the current version, all parameters are in $[0, 1]$. Generating parameters outside that range is easy with the transform $\theta' = \theta/(1-\theta)$. The 4- and 3-layer versions are denoted respectively as `f1` and `g1` in the code.

Model (21) is challenging. I use it as a sandbox to design new optimization techniques. Some have a significant impact on boosting accuracy without increasing the number of epochs, and apply to other models. One not yet tested is to split the input data into multiple bins when n is large, and run each bin separately, in parallel. This is called batch processing in standard DNNs. Bins are based on quantiles of the empirical cumulative distribution function (ECDF). Binning in high dimensions is possible, see chapter 6 in [5], in the context of NoGAN tabular data synthetization including with categorical features.

Like many practical models, (21) is non-identifiable, with different parameter sets leading to the exact same response y. Depending on the parameters, the centers can appear distinctly making this model and its multivariate generalization a good candidate for clustering. At the other extreme, parameters generating too much smoothness (large offsets in particular) result in the m kernels blending together to form a bell curve, a consequence of the central limit theorem. This case is the easiest to handle and works well even with large m, but it is the least useful. For examples of responses, see Figures 11, 12, and 13. The first one has added noise. Good parameterization is key here. Also, it helps to have a rough idea for the offset and skewness parameters, to set good initial conditions.

In the code, the constant s in (21) is denoted as `'small'` and all parameters are in $[0, 1]$. I use swarm optimization for the gradient descent, starting with `ntrials` initial locations called particles, in the parameter space. For each particle I test `subtrials` nearby satellite locations, keeping the best one at each epoch. The global parameter estimate is the best (with minimum loss) among the particles, at each epoch.

D. Large language models

In the context of enterprise large language models (LLMs), it is more efficient to use small token lists specific to a corpus, resulting in millions rather than billions of embeddings, yet reducing hallucinations, increasing accuracy and relevancy, with significant gains in training time. In my next-gen LLMs (called xLLM), I use different types of multi-tokens: regular

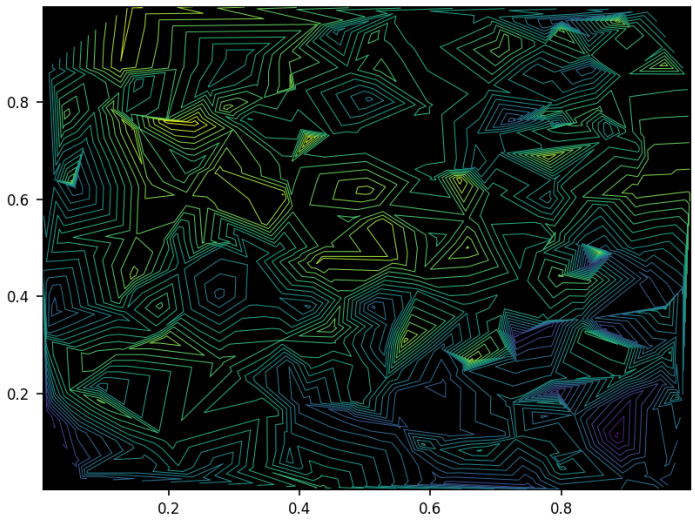

Fig. 3. Model (14), response contour map, first 2 coordinates

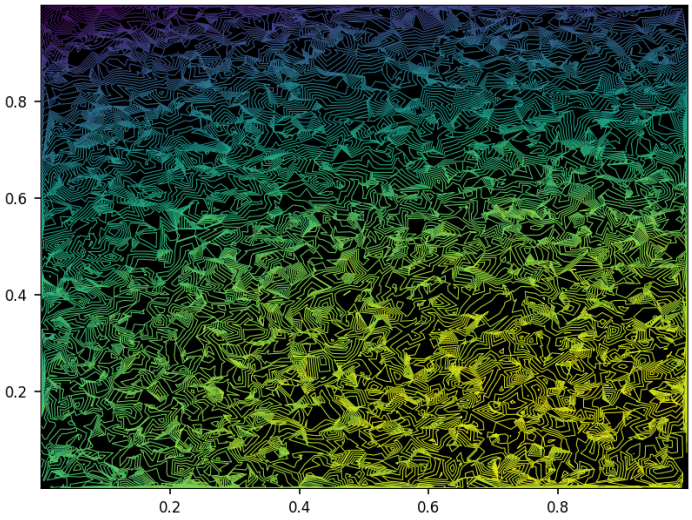

Fig. 4. Same as Figure 3 with yet a different parameter set

or contextual multi-tokens, based on the corpus or augmented with synonyms & acronyms dictionaries or unstemmed lists. Some come from contextual elements such as tags, added post-crawling to each text chunk using a labeling algorithm. Also, I work with variable-length embeddings. Tokens get assigned a different weight depending on type. For instance, context-based multi-tokens are more important than those coming from regular text. See chapter 1 in [3].

For next-token prediction (to turn a structured response to a prompt into long text) the model discussed in II-A is a good candidate. The methodology follows the approach discussed in [8]. In that article, the authors use support vector machines (SVM) as equivalent to DNNs to prove the results: section 2 entitled "Problem Setup" is a good starting point. In my case, I use non-standard DNNs instead of SVMs. This is still a work in progress. For my earlier work on next-token prediction in the context of DNA sequence synthetization, see section 7.1 in [4]. For the connection between DNNs and SVMs, see [1].

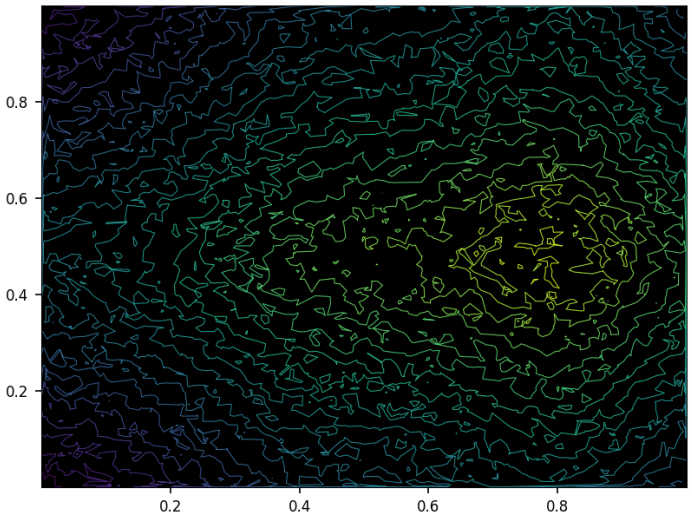

Fig. 5. Same as Figure 3 with different parameter set

III. Model evaluation

Before digging into specific examples, I start with general comments. All examples involve $n = 300$ observations and 500 epochs. The small number of observations is a typical batch size in standard DNNs with batch processing. In all cases, I added the same amount of noise to all synthetic data by setting $\alpha = 0.10$; no noise corresponds to $\alpha = 0$. In some cases, I train the system on 50% of the data by setting the distillation rate to 0.50. I use the loss function L to evaluate the performance at each epoch on the training set (the distilled data). A the end, I also compute the loss on the full (non distilled) data. The ratio L distilled divided by L non distilled is a good indicator of overfitting. It ranges from close to zero (worst) to close to one (best).

I also produce a scatterplot of y (the observed response) vs y_{pred} (the predicted response), see e.g. Figure 7. Finally, I show the correlation between y and y_{pred}. This metric is equivalent to R-squared but overoptimistic since it fails to

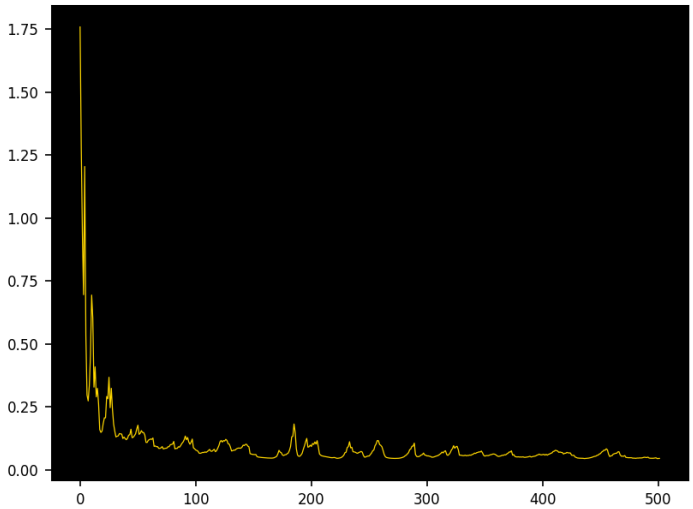

Fig. 6. Loss (Y-axis) over successive epochs (X-axis), case featured in Fig. 3

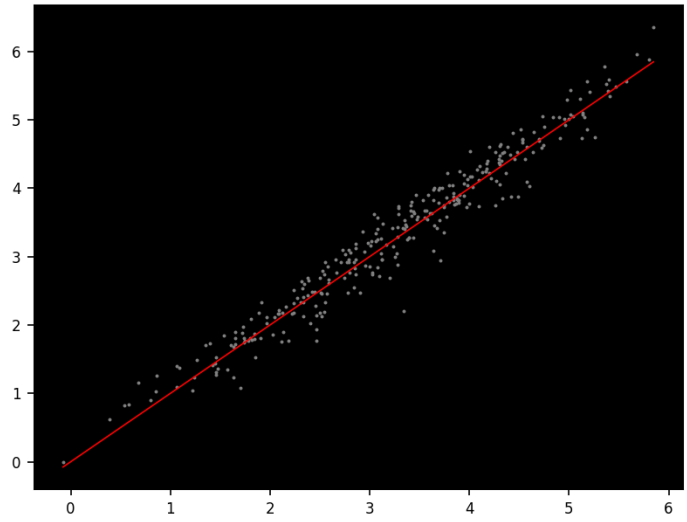

Fig. 7. Observed versus predicted y, case featured in Fig. 3

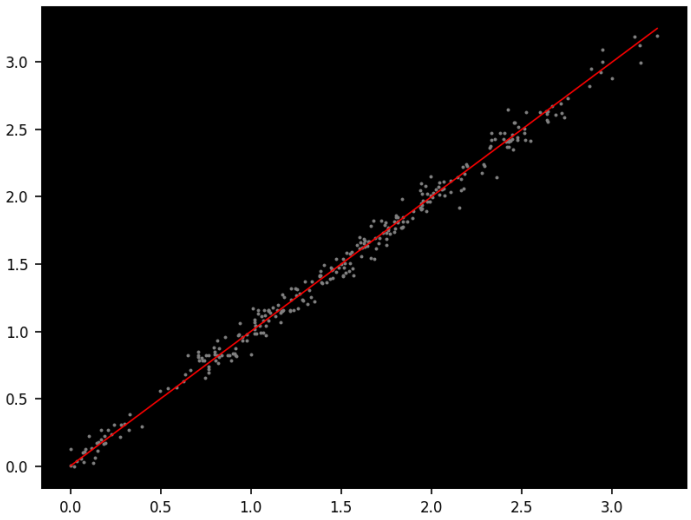

Fig. 8. Observed versus predicted y, model (18)

capture subtle departures from a great fit. In the scatterplot y vs y_{pred}, a good fit results in the dots concentrated around the main diagonal: the red line in Figure 7. If the dots are not randomly distributed around the diagonal, but appeared scattered around some curve, it means the the model fails to capture some patterns and could be improved by adding a layer. This issue may result from heteroskedasticity.

To assess the progress during the descent, I produce a history plot such as Figure 6. The Y-axis represents the loss function; the X-axis is the time (the epoch numbers). It shows when the loss stops gets stuck, and about other potential issues in the gradient descent.

I now discuss the three case studies. The figures in this paper represent a small subset of all the optimizations techniques tested. Not the best, but the ones most representative requiring few epochs and exhibiting increased stability to small changes and generalizations. Zoom in for higher resolution.

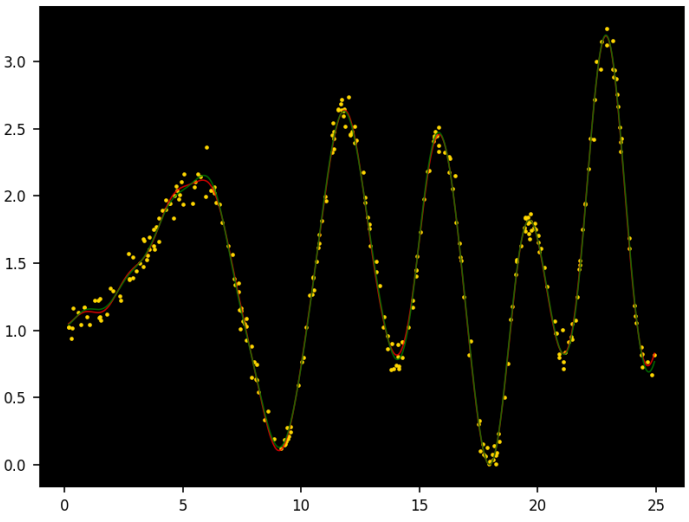

Fig. 9. Data (x, y) as yellow dots, model (18) in red, and predicted response in green

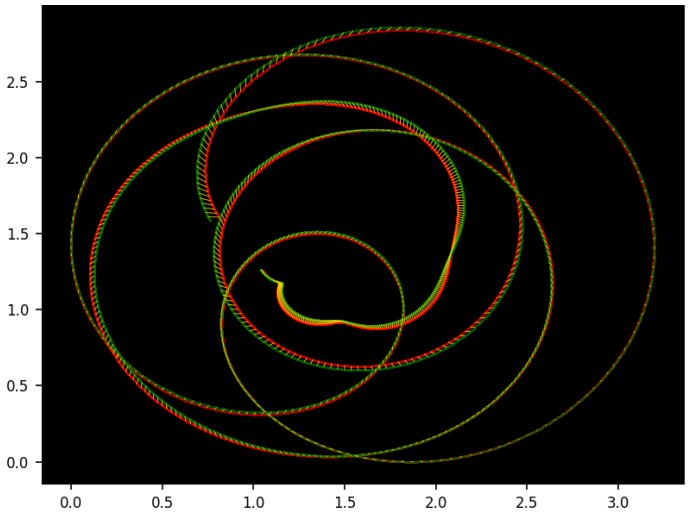

Fig. 10. Orbits linked to model (18) – (19), real in red, predicted in green

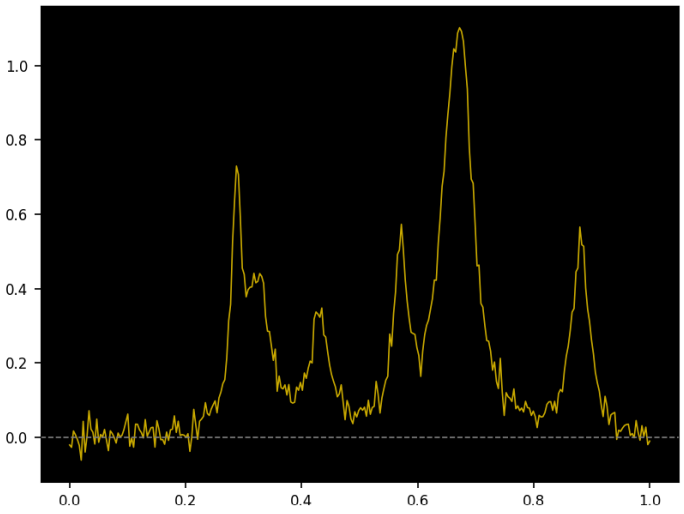

Fig. 11. Model (21) with noise; y on Y-axis, x on X-axis

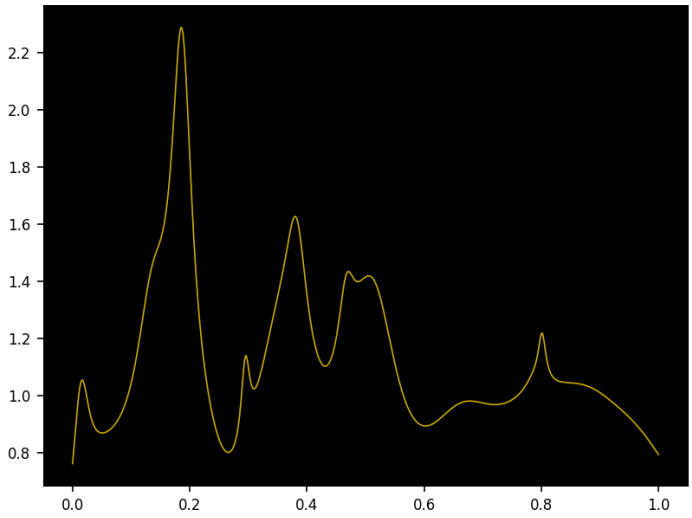

Fig. 12. Model (21) this time with different parameter set, no noise

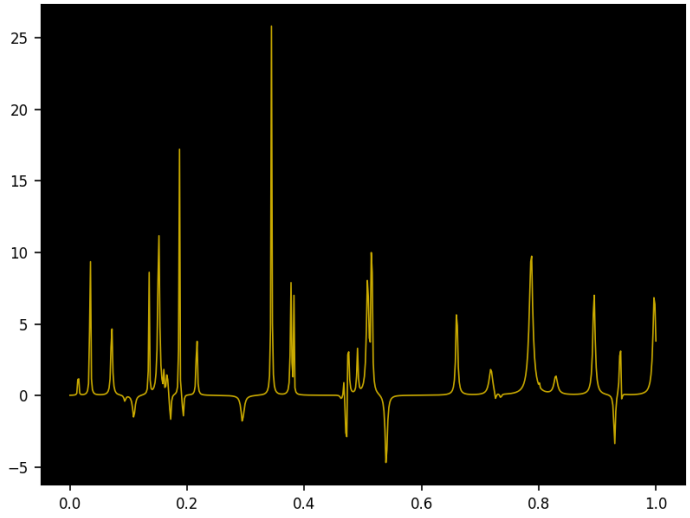

Fig. 13. Model (21) with yet another parameter set, no noise

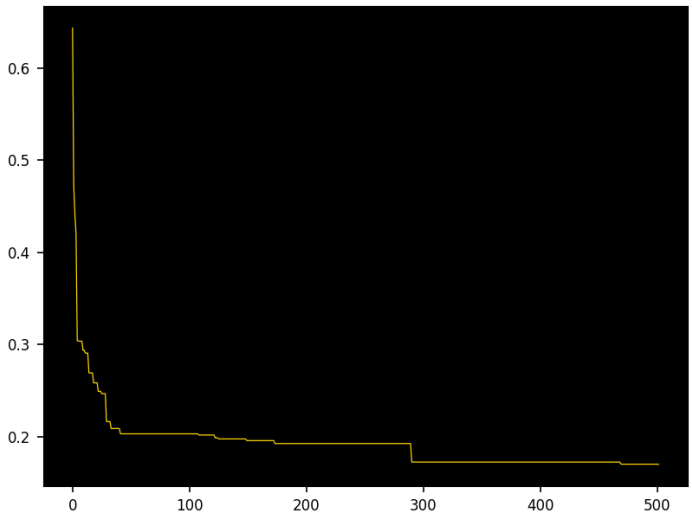

Fig. 14. Loss (Y-axis) over successive epochs (X-axis), model (21) from Figure 11

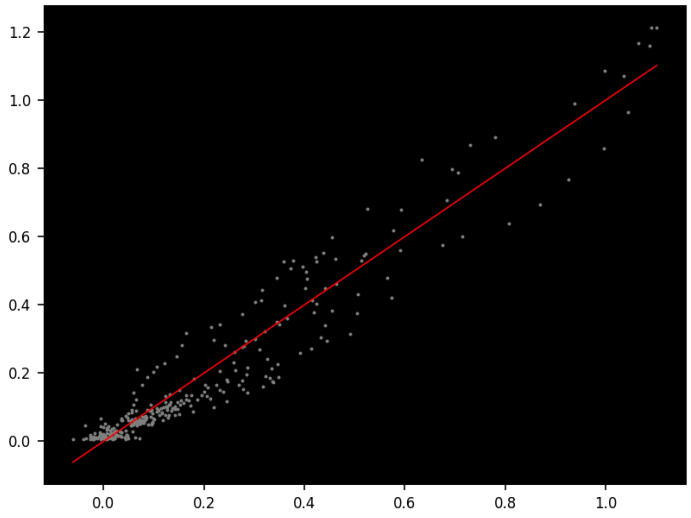

Fig. 15. Observed versus predicted y, model (21) from Figure 11

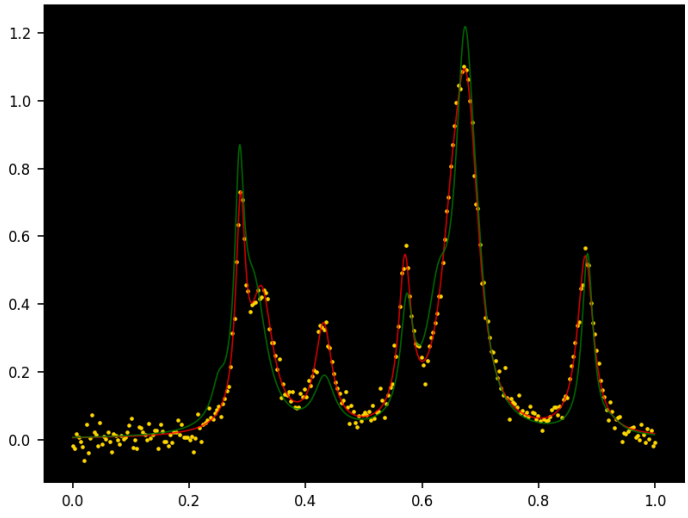

Fig. 16. Data (x, y) as yellow dots, model (21) from Figure 11 in red, and predicted response in green

A. *Model with multidimensional input data*

I tested models (13) and (14) with various θ for the true parameter matrix, with and without the temperature hyperparameter, under L_2 and L_1 descents, with `distillation_rate` set to 0.5 and `alpha` (the noise) set to 0.10. Different optimization techniques compare differently depending on the parameter: there is no single winner, but instead some rules of thumb that you discover with practice.

I also tried different batch sizes n, different learning rates, different thresholds for distillation and for the noise, as well as fine-tuning the hyperparameter `eps` in partial derivatives. The example presented here is the same as in the Python code with the `f0` function, featuring model (14) without temperature. The dimensions retained here are as follows: 3 sub-layers, 5 top layers (thus 15 layers total) and $m = 7$ features (also called dimensions) in the input data x. The response y is 1-D. In its present form, this system is appropriate for curve fitting

in high dimensions, and predictive analytics. Model (13) was tested with over 10^3 parameters. As with standard DNNs, these models can be adapted to other problems.

I haven't optimized the speed yet. For instance, storing the parameter as an interlaced 2-D matrix rather than a 3-D tensor, could be suboptimal. Also, changing the summation order in double summations may improve the speed. Finally, I want to test the impact of optimizing one layer at a time within each epoch, rather than globally as of now. Figures 6 and 7 show the performance for my specific case. The former includes temperature decay, explaining the bumpy loss.

B. An interesting universal function

One of the best kept secrets for modeling any response, model (18) is easy to work with. Applications include signal processing, engineering, and astronomy as shown in Figure 10 Illustrated here with 3 layers and 90 parameters (functions `f2` and `f3` in the code): see Figures 8 and 9 for performance.

Next steps: generalize to higher dimensions for the input data, further test its extrapolation capabilities, and analyze the shape of the basin of attraction in the parameter space: values of the initial parameters that lead to fast convergence. In dynamical systems, basins of attractions usually have fractal-like boundaries. In the end, the gradient descent iterations form a dynamical system.

C. Example with singularities

In terms of testing, much of what I discussed in section III-A also applies to this case. Here, despite dealing with a function f differentiable everywhere with respect to all parameters except when the skewness or offset is zero (the lower bound in the parameter space), the L_1 approach provides a better fit compared to L_2, especially around the origin. This case involves models (21) and (9), that is, `f1` and `g1` in the code. It also illustrates reparameterization and swarm optimization. The former is discussed in section I-C. Based on observed results, it confirms the fact that adding redundancy in the parameters (adding an unnecessary extra layer) can improve the fit.

In swarm optimization, the speed depends on the product $n_t \times n_s$ where n_t, n_s are the number of trials (starting points) and subtrials (branching points per trial). Extreme values such as $(n_t, n_s) = (1, 100)$ work as well as mid-range, say $(10, 10)$. Finally, I added fictitious parameters (also called ghost parameters) to boost accuracy, by setting some weights to zero in the real (unknown) parameter set, and increasing the number m of features accordingly. Another way to describe it is using overparameterization.

Performance is shown in Figures 14, 15 and 16 with 4 layers and 48 parameters, without temperature. The loss measured on distilled data (with 50% distillation) is 0.17 vs 0.19 on the full data. Since both numbers are similar, overfitting is not an issue. Figure 16 shows that the fit, while not bad, is not as good as the 0.96 correlation between y and $y_{\texttt{pred}}$ makes it to be. This is further confirmed by looking at Figure 15. Thus, R-squared (the square of the correlation) is a poor evaluation metric for this type of spiky data.

In Figure 15, the response y has values below zero because of added noise. My system can not produce negative values with positive weights. You can see it by looking at a number of $y_{\texttt{pred}}$ values stacked at zero. It is not a defect but a quality, since this model is for positive responses. The issue is in the data, not the predictions! One way to correct for this is to use a multiplicative rather than additive noise when synthesizing the input data. It may introduce a multiplicative bias, that is easily corrected in a post-processing step via a linear transform. More insights below:

- Can you represent any positive response (other than full chaos, pattern-free) with a combination of kernels and parameters in $[0, 1]$ as discussed in this article? The answer is yes. If anything, with the transform $\theta' = \theta/(1-\theta)$. Parameters in $[-1, 1]$ or adding a negative offset can take care of responses of mixed sign.
- Designing DNNs is like building a chair. You add more screws than needed, not tightened, allowing for leeway. Otherwise, you cannot finish your chair: it is like a classic ML model with few parameters, tightly fit but lacking adaptability.
- If using temperature or stochastic descent, do not accept parameter updates if the updated loss is > 1.5 times the current loss (assuming positive loss function). Chances are that it won't be the case at the next epoch. Also, save the best parameter set obtained during descent; it may not be the one obtained at the last epoch. Or use some other stopping rule.
- Start with good parameter estimates. Easier with intuitive parameters. Or use a preprocessing step to get decent guesses. Bin the input data and get θ estimates for each bin: this is called batch processing. Try various reparameterization schemes, some with more layers.

IV. Conclusions

This original non-standard DNN architecture turns black boxes into explainable AI with intuitive parameters. The focus is on speed (fewer parameters, faster convergence), robustness, and versatility with easy, rapid fine-tune and many options. It does not use PyTorch or similar libraries: you have full control over the code, increasing security and customization. Tested on synthetic data batches for predictive analytics, high dimensional curve fitting and noise filtering in various settings, it incorporates many innovative features. Some of them, like the equalizer, have a dramatic impact on performance and can also be implemented in standard DNNs. The core of the code is simple and consists of fewer than 200 lines, relying on Numpy alone.

V. Python code

The program `dnn.py` is also on GitHub, here. Start by looking at the functions `loss`, `partial_derivatives`, `init_L2_loss` and `gradient_descent`.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['axes.facecolor'] = 'black'

#--- 1. Distillation

def distill(rate, x, y):
   n = len(y)
   nd = int(rate * n)
   chosen = np.random.choice(np.arange(0, n),
       size=nd, replace=False)
   if len(x.shape) == 2:
      xd = x[:,chosen] # only keep cols in chosen
   else:
      xd = x[chosen]
   yd = y[chosen]
   print("Distillation factor:", 1-rate)
   return(xd, yd)


#--- 2. Test functions for curve fitting

def f0(params, x, args=""):
   model = args['model']
   nfeatures, nobs = x.shape # nfeatures is m in
       the paper
   layers, lparams = params.shape
   ones = np.ones(nobs)
   z = np.zeros(nobs)
   J = layers // 4
   for j in range(J):
      for k in range(nfeatures):
         theta0 = params[4*j,k]
         theta1 = params[4*j+1,k]*ones
         theta2 = params[4*j+2,k]
         theta3 = 1 # try: 1; params[4*j+3,k];
             1-theta2
         if model == 'approx. gaussian':
            z += theta0*(ones -
                ((x[k,:]-theta1)*theta2)**2)
         elif model == 'gaussian':
            z += theta0*np.exp(-(theta3*(x[k,:] -
                theta1)/theta2)**2)
   if args['equalize']:
      z = z - np.min(z)
   return(z)

def f1(params, x, args = ""):
   small = args['small']
   type = args['function_type']
   xnodes = args['centers']
   model = args['model']
   layers, nfeatures = params.shape
   nobs = len(x)
   ones = np.ones(nobs)
   small_ones = small*ones
   z = np.zeros(nobs)
   for k in range(nfeatures):
      if model == 'static':
         # centers are fixed
         xnode_k = xnodes[k]*ones
      elif model == 'latent':
         # centers must be estimated
         xnode_k = params[3,k]*ones
      if type == 'rational':
         z +=
             small*params[0,k]/(params[1,k]*small_ones
             +((x-xnode_k)/params[2,k])**2)
      elif type == 'polynom':
         z += params[0,k]*(x-xnode_k)+
             params[1,k]*(x-xnode_k)**2
             +params[2,k]*(x-xnode_k)**3
   if args['equalize']:
      z = z - np.min(z)
   return(z)

def f2(params, x, args=""):
   # Dirichet eta function, real part with fixed
       sigma
   layers, nfeatures = params.shape
   sigma = params[0, 0]
   phi = params[1, 0]
   theta = phi/(1-phi)
   z = 0
   for k in range(nfeatures):
      z += params[2, k] * (-1)**k *
          np.cos((x+theta)*np.log(k+1)) /
          (k+1)**(sigma)
   if args['equalize']:
      z = z - np.min(z)
   return(z)

def f3(params, x, args=""):
   # Dirichet eta function, imaginary part with
       fixed sigma
   layers, nfeatures = params.shape
   sigma = params[0, 0]
   phi = params[1, 0]
   theta = phi/(1-phi)
   z = 0
   for k in range(nfeatures):
      z += params[2, k] * (-1)**k *
          np.sin((x+theta)*np.log(k+1)) /
          (k+1)**(sigma)
   if args['equalize']:
      z = z - np.min(z)
   return(z)

def g1(rparams, x, args=""):
   small = args['small']
   type = args['function_type']
   xnodes = args['centers']
   model = args['model']
   layers, nfeatures = rparams.shape
   nobs = len(x)
   z = np.zeros(nobs)
   ones = np.ones(nobs)
   small_ones = small*ones
   for k in range(nfeatures):
      if model == 'static':
         # centers are fixed
         xnode_k = xnodes[k]*ones
      elif model == 'latent':
         # centers must be estimated
         xnode_k = rparams[2,k]*ones
      z += small*rparams[0,
          k]/(small_ones+((x-xnode_k)/rparams[1,
          k])**2)
   if args['equalize']:
      z = z - np.min(z)
   return(z)


#--- 3. Reparameterization

def reparameterize(params):
   # number of layers in rparams must be <= to that
       in params
   layers, nfeatures = params.shape
   rparams = np.zeros((layers, nfeatures))
   for k in range(nfeatures):
      rparams[0, k] = params[0, k]/params[1, k]
      rparams[1, k] = np.sqrt(params[1,
          k])*params[2, k]
      rparams[2, k] = params[3, k]
   return(rparams)

def check_reparametrization(f, g, params, x,
    args=""):
   y_from_f = f(params, x, args)
   y_from_g = g(reparameterize(params), x, args)
```

```
   delta = np.max(abs(y_from_f - y_from_g))
   if delta > 0.000001 or np.isnan(delta):
      print("reparameterization error: delta =",
          delta)
      exit()
   else:
      print("reparameterization correct: delta =",
          delta)
   return(delta)


#--- 4. Gradient descent

def loss(f, params_estimated, y, x, mode, args=""):

   y_estimated = f(params_estimated, x, args)
   z = np.abs(y - y_estimated)
   if mode == 'L1_abs':
      return(np.max(z))
   elif mode == 'L1_avg':
      return(np.average(z))
   elif mode == 'L2':
      return(np.dot(z, z)/len(z))

def init_L2_descent(f, y, x, L_error, layers,
    nfeatures, args):

    params_init = np.full((layers, nfeatures), 0.5)
    init_loss = loss(f, params_init, y, x, L_error,
        args)
    args['params_init'] = params_init
    args['init_loss'] = init_loss
    return(init_loss, args)

def init_swarm_descent(f, y, x, L_error, layers,
    nfeatures, args):

   ntrials = args['ntrials'] # trial called
       particle in litterature
   loss_swarm = np.zeros(ntrials)
   min_loss = 99999999.99
   params_swarm = []
   for k in range(ntrials):
      params_init = np.zeros((layers, nfeatures))
      params_init[0,:] = np.random.uniform(0.00,
          0.50, nfeatures) # weights
      params_init[1,:] = np.random.uniform(0.25,
          1.00, nfeatures) # offsets
      params_init[2,:] = np.random.uniform(0.25,
          1.00, nfeatures) # skewness
      params_init[3,:] = np.random.uniform(0.00,
          1.00, nfeatures) # centers
      params_swarm.append(params_init)
      loss_val = loss(f, params_init, y, x,
          L_error, args)
      loss_swarm[k] = loss_val
      args['params_swarm'] = params_swarm
      args['loss_swarm'] = loss_swarm
      if loss_val < min_loss:
         min_loss = loss_val
         params_best = np.copy(params_init)
   args['params_init'] = np.copy(params_best)
   args['params_estimated'] = np.copy(params_best)
   return(np.min(loss_swarm), args)

def swarm_descent(loss, f, params_estimated, y, x,
    learning_rate, temp, mode, args=""):

   # trial (or starting point) is sometimes called
       particle in swarm optimization
   ntrials = args['ntrials']
   subtrials = args['subtrials']
   params_swarm = args['params_swarm'] # one
       parameter set per trial
   loss_swarm = args['loss_swarm'] # one loss value
       per trial
   L_current = args['current_loss']
   L_best_global = L_current
   norm = np.sum(abs(f(params_estimated,x,
       args)))/len(y)
   ones = np.full((layers, nfeatures), 1)

   for trial in range(ntrials):
      L_best = loss_swarm[trial]
      for k in range(subtrials):
         pd = np.random.uniform(-1, 1, (layers,
             nfeatures)) * norm
         params_test = params_swarm[trial] -
             learning_rate * pd
         params_test = np.abs(params_test)
         L_new = loss(f, params_test, y, x, mode,
             args)
         if L_new < L_best or np.random.uniform() <
             temp:
            L_best = L_new
            params_swarm[trial] =
                np.copy(params_test)
            loss_swarm[trial] = L_best
      if loss_swarm[trial] < L_best_global:
         L_best_global = loss_swarm[trial]
         params_estimated = params_swarm[trial]
   return(params_estimated)

def partial_derivatives(loss, f, params_estimated,
    y, x, learning_rate, temp, L_error, args=""):

   # partial derivatives (pd) of loss function with
       respect to
   #   the coefficients in params_estimated; use it
       if mode='L2'

   eps = args['eps']
   layers, nfeatures = params_estimated.shape
   pd = np.zeros((layers, nfeatures))
   for l in range(layers):
      for k in range(nfeatures):
         params_eps = np.copy(params_estimated)
         # make eps depend on epoch, layer,
             parameter ?
         params_eps[l, k] += eps
         L_right = loss(f, params_eps, y, x,
             L_error, args)
         L_current = args['current_loss']
         pd[l, k] = (L_right - L_current) / eps
         if np.random.uniform() < temp:
            pd[l, k] = -pd[l, k]
   params_new = params_estimated - learning_rate *
       pd
   for index, value in
       np.ndenumerate(params_estimated):
      if 0 < value < 1:
         # handle situation where params is out of
             accepted range
         params_estimated[index] = params_new[index]
   return(params_estimated)

def gradient_descent(epochs, learning_rate, layers,
    nfeatures, f, y, x, temperature, L_error,
    descent, args=""):

   n = len(y)
   epoch = 0
   history = []
   if descent == 'L2':
      (mae, args) = init_L2_descent(f, y, x,
          L_error, layers, nfeatures, args)
   elif descent == 'swarm_descent':
      (mae, args) = init_swarm_descent(f, y, x,
          L_error, layers, nfeatures, args)
   history.append(mae)
   args['current_loss'] = history[-1]
   params_estimated = args['params_init']

   while epoch < epochs:
```

```
      decay = history[-1]
      temp = temperature * decay
      if descent == 'swarm_descent':
         params_estimated = swarm_descent(loss, f,
             params_estimated, y, x,
                              learning_rate, temp,
                                  L_error, args)
      elif descent == 'L2':
         params_estimated =
             partial_derivatives(loss, f,
             params_estimated, y, x,
                              learning_rate, temp,
                                  L_error, args)
      mae = loss(f, params_estimated, y, x,
          L_error, args)
      history.append(mae)
      args['current_loss'] = history[-1]
      if epoch % 100 == 0:
         print("Epoch %5d MAE %8.5f" %(epoch, mae))
      epoch += 1
   return(params_estimated, history)


#--- 5. Visualization

def summary(x, y, params, params_estimated, f,
    L_error, args):

   y_estimated = f(params_estimated, x, args)
   visu(history, y, x, f, params_estimated, params,
       args)
   mae_full_data = loss(f, params_estimated, y, x,
       L_error, args)

   corr = abs(np.corrcoef(y, y_estimated))[0, 1]
   print("mae_d: %8.5f mae_f: %8.5f corr: %8.5f
       temp: %6.4f function: %s (n=%5d)"
          % (history[-1], mae_full_data, corr,
              temperature, 'f', len(y)))
   return()

def visu(history, y, x, f, params_estimated,
    params, args):

   xvalues = np.arange(len(history))
   plt.plot(xvalues, history, linewidth = 0.6,
       c='gold')
   plt.show()
   y_estimated = f(params_estimated, x, args)
   plt.scatter(y, y_estimated, s = 0.8, c='gray')
   ymin = np.min(y)
   ymax = np.max(y)
   plt.plot((ymin, ymax), (ymin, ymax), c='red',
       linewidth = 0.8, alpha=1.0)
   plt.show()
   if len(x.shape) == 1:
      # input data is 1-Dim
      xvalues = np.arange(np.min(x), np.max(x),
          0.001)
      y_base = f(params, xvalues, args)
      y_reconstructed = f(params_estimated,
          xvalues, args)
      plt.plot(xvalues, y_base, c='red', linewidth
          = 0.8, alpha = 0.8)
      plt.plot(xvalues, y_reconstructed, c='green',
          linewidth = 0.8, alpha = 0.8)
      plt.scatter(x, y, c='gold', s=1.5)
      plt.show()
   return()

def visu_eta_orbit(f2, f3, x, y, params,
    params_estimated, args):

   xmin, xmax = np.min(x), np.max(x)
   step = (xmax - xmin)/ 5000
   xvalues = np.arange(xmin, xmax, step)
   y_exact_real = f2(params, xvalues, args)
   y_reconstructed_real = f2(params_estimated,
       xvalues, args)
   y_exact_imaginary = f3(params, xvalues, args)
   y_reconstructed_imaginary = f3(params_estimated,
       xvalues, args)
   y_estimated_imaginary = f3(params_estimated, x,
       args)
   plt.scatter(y_exact_real, y_exact_imaginary,
       s=0.08, c='red')
   plt.scatter(y_reconstructed_real,
       y_reconstructed_imaginary, s=0.08, c='green')
   for k in range(len(xvalues)):
      if k %5 == 0:
         exact = (y_exact_real[k],
             y_exact_imaginary[k])
         reconstructed = (y_reconstructed_real[k],
             y_reconstructed_imaginary[k])
         plt.plot((exact[0], reconstructed[0]),
             (exact[1],reconstructed[1]), c='gold',
             linewidth = 0.4)
   plt.show()
   return()


#--- 6. Initializations

#-- 6.1. default parameters

# L_error: options for model evaluation: 'L1_abs',
    'L1_avg', 'L2'
# descent: options for type of descent: 'L1_abs',
    'L1_avg', 'L2'

n = 300          # number of observations
nfeatures = 10   # number of features
seed = 565       # for replicability
eps = 0.000001   # precision on partial derivatives
alpha = 0.10     # amount of noise to add to y
epochs = 501     # iterations in gradient descent
layers = 3       # number of parameter subsets
learning_rate = 0.1 # learning rate
temperature = 0  # minimum is 0; large value =>
    more entropy in descent
L_error = 'L2'   # 'L2' minimizes MSE loss,
    'L1_abs' minimizes MAE loss
descent = 'L2'   # descent algorithm: 'L1_abs' or
    'L2' (best: descent=L_error)
distill_rate = 1.0 # proportions of obs left in
    data after random deletion
args = {}        # list of hyperparameters passed
    across all functions
args['eps'] = eps # for 'L2' descent only
np.random.seed(seed)


# choose type of function
f = f2  # choices: f0, f1, f2

#-- 6.2. parameters & arguments for each function
    type

if f == f0: # multivariate curve fitting;
    dim=nfeatures; multiple layers

   nfeatures = 7
   centers = 5     # top layers, better if smaller
       than nfeatures
   layers = 4*centers # here 4 is the max number of
       sub-layers per top layer
   distill_rate = 0.5
   params = np.random.uniform(0, 1, (layers,
       nfeatures))
   x = np.random.uniform(0, 1, (nfeatures, n))
   args['equalize'] = True
   args['model'] = 'gaussian' # 'approx. gaussian'
       or 'gaussian'
```

```
elif f == f1: # curve fitting in 1D, multiple
    layers; chaotic + singularities

   nfeatures = 12 # the number of centers in the
       parameter space
   L_error = 'L2'
   descent = 'swarm_descent'
   layers = 4
   params = np.zeros((layers, nfeatures))
   params[0,:] = np.random.uniform(0.25, 0.50,
       nfeatures) # weights
   params[1,:] = np.random.uniform(0.25, 1.00,
       nfeatures) # offset
   params[2,:] = np.random.uniform(0.25, 1.00,
       nfeatures) # skewness
   params[3,:] = np.random.uniform(0.00, 1.00,
       nfeatures) # centers
   params[0,:4] = [0, 0, 0 , 0] # first 4 features
       not used for f, only for DNN
   x = np.linspace(0, 1, num=n) # generate
       observations
   args['equalize'] = False
   args['small'] = 0.001      # > 0; singularity if
       small=0
   args['function_type'] = 'rational' # 'rational'
       or 'polynom'
   args['centers'] = params[3, :] # used as fixed
       centers in static model
   args['model'] = 'latent'   # 'latent' (moving
       centers) or 'static'
   args['ntrials'] = 4 ## 2 #10 # for
       descent='swarm_descent'; large ==> fewer
       epochs
   args['subtrials'] = 50 ## 100 #10 # number of
       particles in 'swarm_descent' descent

elif f == f2: # Using Riemann zeta function to
    model and predict orbits

   nfeatures = 30
   params = np.zeros((layers, nfeatures))
   params[0, 0] = 0.75
   params[1, 0] = 0.10
   for k in range(nfeatures):
      params[2, k] = 1
   x = np.random.uniform(0, 25, n)
   args['equalize'] = True

#-- 6.3. distillation, and adding noise to response
    y

y_base = f(params, x, args)      # generate base
    response
y = np.copy(y_base)
stdev_y = np.std(y)              # standard
    deviation of response
y += np.random.normal(0, alpha*stdev_y, n) # add
    noise to response
(xd, yd) = distill(distill_rate, x, y) # work on
    sub-sample of x, y
np.random.seed(seed)             # reset seed
    post-distillation

if f == f0 and nfeatures >= 2:
   plt.tricontour(x[0,:], x[1,:], y, levels=20,
       cmap='viridis', linewidths=0.5) ##100 levels
   plt.show()
elif f == f1:
   plt.plot(x, y, c='gold', linewidth = 0.8, alpha
       = 0.8)
   plt.axhline(y = 0, color='gray', linewidth =
       0.8, linestyle='--')
   plt.show()


#--- 7. Main: without reparameterization

temperature = 0.00
(params_estimated, history) =
    gradient_descent(epochs, learning_rate, layers,
    nfeatures, f, yd, xd, temperature, L_error,
    descent, args)
summary(x, y, params, params_estimated, f, L_error,
    args)
if f == f2:
   visu_eta_orbit(f2, f3, x, y, params,
       params_estimated, args)

temperature = 0.30
(params_estimated, history) =
    gradient_descent(epochs, learning_rate, layers,
    nfeatures, f, yd, xd, temperature, L_error,
    descent, args)
summary(x, y, params, params_estimated, f, L_error,
    args)
if f == f2:
   visu_eta_orbit(f2, f3, x, y, params,
       params_estimated, args)

#--- 8. Main: with reparameterization

if f == f1:

   # g1 = f1 but reparameterized, using 3 layers
       rather than 4
   check_reparametrization(f1, g1, params, x, args)
       # exit if incorrect reparameterization
   rparams = reparameterize(params)

   temperature = 0.00
   (rparams_estimated, history) =
       gradient_descent(epochs, learning_rate,
       layers, nfeatures, g1, yd, xd, temperature,
       L_error, descent, args)
   summary(x, y, rparams, rparams_estimated, g1,
       L_error, args)
```

References

[1] Yilan Chen et al. On the equivalence between neural network and support vector machines. *Proceedings of the 35th Conference on Neural Information Processing Systems*, pages 1–13, 2021. NeurIPS 2021 [Link]. 7

[2] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [Link]. 5

[3] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLTechniques.com, 2024. [Link]. 2, 7

[4] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions*. MLTechniques.com, 2024. [Link]. 3, 7

[5] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLTechniques.com, 2024. [Link]. 4, 5, 6

[6] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 3, 5, 6

[7] Vincent Granville. *0 and 1 – From Elemental Math to Quantum AI*. MLTechniques.com, 2025. [Link]. 5

[8] Yingcong Li et al. Mechanics of next token prediction with self-attention. *Proceedings of the 27th International Conference on Artificial Intelligence and Statistics*, pages 1–43, 2024. arXiv:2403:08081 [Link]. 7

Vincent Granville Vincent Granville is a pioneering GenAI scientist, co-founder at BondingAI.io, the LLM 2.0 platform for hallucination-free, secure, in-house, lightning-fast Enterprise AI at scale with zero weight and no GPU. He is also author (Elsevier, Wiley), publisher, and successful entrepreneur with multi-million-dollar exit. Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET. He completed a post-doc in computational statistics at University of Cambridge.

INDEX