Jacobian-Enhanced Neural Network

Steven H. Berguin[*]

Georgia Tech Research Institute (GTRI), Atlanta, GA, 30318

April 21, 2024

Executive Summary

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Nomenclature

L	number of layers in neural network
a	activation output, $a = g(z)$
b	bias (parameter to be learned)
g	activation function
m	number of training examples
n	number of nodes in a layer
n_x	number of nodes in input layer, $n_x = n^{[0]}$
n_y	number of nodes in output layer, $n_y = n^{[L]}$
w	weight (parameter to be learned)
x	training data input
y	training data output
$\hat{y}$	predicted output
z	linear activation input

Hyperparameters

α	learning rate
β	hyperparameter to priorities training data values
γ	hyperparameter to priorities training data partials
λ	hyperparameter used for regularization

Mathematical Symbols and Operators

$\mathcal{J}$	cost function
$\mathcal{L}$	loss function
$()_{j'}$	first derivative, $\partial()/\partial x_j$
$()_{j''}$	second derivative, $\partial^2()/\partial x_j^2$

Superscripts/Subscripts

[*]Senior Research Engineer, Electronics System Laboratory (ELSYS) / Systems Engineering Research Division (SERD)

(i)	i^{th} training example
i	i^{th} node in layer
j	j^{th} input, x_j
k	k^{th} output, y_k
l	l^{th} layer of neural network
r	r^{th} node of *current* layer l
s	s^{th} node of *previous* layer $l-1$

1 Introduction

Jacobian-Enhanced Neural Networks (JENN) are fully connected multi-layer perceptrons, whose training process is modified to predict partial derivatives accurately. This is accomplished by minimizing a modified version of the Least Squares Estimator (LSE) that accounts for Jacobian prediction error, where a Jacobian extends the concept a gradient for vector functions with more than one output. The main benefit of jacobian-enhancement (synonymous to gradient-enhancement) is better accuracy with fewer training points compared to standard fully connected neural nets. The objective of this paper is to document the theory behind the supporting Python library of the same name and quantify its benefits for the intended use-case of Surrogate-Based Optimization (SBO). This paper differs from previous work in the following ways. First, it provides a concise but complete derivation of the theory, including equations in vectorized form (see appendix), without relying on automatic differentiation. Second, it is a gradient-enhanced, deep-learning formulation that can handle any number of hidden layers, inputs, or outputs. Finally, through judicious use of hyperparameters, it offers an original formulation that uniquely enables individual sample points to be prioritized differently, in situations that call for it. The latter, in particular, gives rise to the concept of *polishing* (described in the application section) which offers a new research avenue to help improve SBO for gradient-based optimization.

1.1 Intended Use-Case

JENN is intended for the field of computer aided design, where there is often a need to replace computationally expensive, physics-based models with fast running approximations, commonly referred to as surrogate models or meta-models. Since a surrogate emulates the original model accurately in real time, it offers a speed benefit that can be used to carry out orders of magnitude more function calls quickly. This opens the door to useful design applications, such as Monte Carlo simulation of computationally expensive functions for example. However, this is true for any type of surrogate model in general. In the special case of gradient-enhanced methods, there is the additional value proposition that partial derivatives are accurate, which is a critical property for one important use-case: surrogate-based optimization. The field of aerospace engineering is rich in design applications of such a use-case, with entire book chapters dedicated to it [1, 2] and gradient-enhanced use-cases typically found in aerodynamic applications [3, 4, 5].

1.2 Audience & Software

The open-source, supporting Python library can be found at `https://pypi.org/project/jenn/`. There exist many excellent deep learning frameworks, such as tensorflow [6], which are more performant and versatile than JENN, as well as many all-purpose machine learning libraries such as scikit-learn [7]. However, gradient-enhancement is not inherently part of those popular frameworks, let alone the availability of prepared functions to predict partials after training is complete, and requires significant effort to be implemented in those tools. The present library seeks to close that gap for neural networks; it is intended for those engineers in a rush with a need to predict partials accurately and who are seeking a minimal-effort Application Programming Interface (API) with low-barrier to entry[1].

[1]Note that the Surrogate Modeling Toolbox [8] is a Python project with the same goal, which uses JENN under the hood, but offers a different API that can be used to switch out their different types of models (present author contribution)

1.3 Limitations

The value proposition of a surrogate is that the computational expense of generating training data to fit the model is much less than the computational expense of performing the analysis with the original physics-based model itself, which is not always true. Gradient-enhanced methods require responses to be continuous and smooth, but they are only beneficial if the cost of obtaining partials is not excessive in the first place (e.g. adjoint methods [9] in computational fluid dynamics), or if the need for accuracy outweighs the cost of computing the partials. One should therefore carefully weigh the benefit of gradient-enhanced methods relative to the needs of their application.

2 Previous Work

A cursory literature research reveals that the field of gradient-enhanced regression is predominantly focused on Gaussian processes. Dedicated book chapters can be found about gradient-enhanced Kriging models [2], popular Python libraries have been almost exclusively devoted to gradient-enhanced Bayesian methods [8], and even review papers about gradient-enhanced meta-models do not mention neural networks, except in passing [10]. However, early work from the late 1990's can be found [11] which uses gradient-enhanced neural network approximations in the context of Concurrent Subspace Optimization (CSSO), a type of optimization architecture in the field of multi-disciplinary design optimization. Those authors conclude that gradient-enhanced neural networks are more accurate and require significantly less training data than their non-enhanced counterparts, which seems to be the general consensus in all work surveyed for this paper.

Follow-on research by that group provides the mathematical derivation of the equations for neural net gradient-enhancement [12], but the formulation is limited to shallow neural networks with only two layers, which is of limited utility. This shortcoming is overcome in the late 2000's by a more general formulation for deep neural networks [3], where the authors take advantage of improved neural net accuracy to enable surrogate-based optimization of airfoil designs using evolution algorithms. All necessary equations to reproduce their algorithm are provided, but the vectorized form for computer programming is missing and their formulation is limited to only one output.

Recent work in 2019 takes a different approach, relying on established deep-learning frameworks to skip back-propagation in favor of automatic differentiation [4]. Those authors successfully apply gradient-enhanced neural networks to aerodynamic use-cases and re-affirm the previous conclusion that such techniques improve accuracy and reduce the amount of training data needed, provided the cost of obtaining the gradient is not counter-productive in the first place. While deep learning frameworks have obvious advantages (*e.g.* scalability), there is still complementary research value in developing the equations for back-propagation, which is a special case of automatic differentiation for scalar function. In particular, stand-alone equations are reproducible without third-party tools and convey a deeper understanding of the mechanics behind an algorithm, which others can build upon.

Finally, the most recent work on gradient-enhanced neural nets focuses on so called Physics-Informed Neural Networks [13] (PINN). This creative research uses the residuals of Partial Differential Equations (PDE) governing some physical system as the loss function for training neural nets, instead of the typical least squares estimator for example. However, PINN is outside the target scope of the present work, which seeks to develop a general purpose algorithm to fit any smooth function, regardless of what it represents, even if it is not physics-informed.

3 Mathematical Derivation

Jacobian-Enhanced Neural Networks (JENN) follow the same steps as standard Neural Networks (NN), shown in Fig. 1, except that the cost function used during parameter update is modified to account for partial derivatives, with the consequence that back propagation must be modified accordingly. This section provides the mathematical derivation for Jacobian-enhancement.

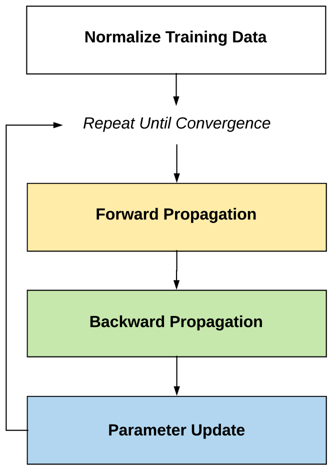


Figure 1: Summary of neural net training process

3.1 Notation

For clarity, let us adopt a similar notation to the one used by Andrew Ng [?] where superscript $[l]$ denotes the layer number, (i) is the training example, and subscript i indicates the node number in a layer. For instance, the activation of the i^{th} node in layer $[l]$ for example (i) would be denoted:

$$a_i^{[l](i)} = g\left(z_i^{[l](i)}\right) \quad \forall \quad i \in [1, n^{[l]}] \quad (i) \in [1, m] \quad [l] \in [1, L] \tag{1}$$

To avoid clutter in the derivation to follow, it will be understood that subscripts r and s refer to quantities associated with the current and previous layer, respectively, evaluated at example (i) without the need for superscripts. Concretely:

$$a_r \equiv a_{i=r}^{[l](i)} \quad \text{and} \quad z_r \equiv z_{i=r}^{[l](i)} \quad \forall \quad r \in [1, n^{[l]}] \tag{2}$$

$$a'_{jr} \equiv \frac{\partial a_r^{[l]}}{\partial x_j}^{(i)} \quad \text{and} \quad z'_{jr} \equiv \frac{\partial z_r^{[l]}}{\partial x_j}^{(i)} \quad \forall \quad r \in [1, n^{[l]}] \tag{3}$$

$$a_s \equiv a_{i=s}^{[l-1](i)} \quad \text{and} \quad z_s \equiv z_{i=s}^{[l-1](i)} \quad \forall \quad s \in [1, n^{[l-1]}] \tag{4}$$

$$a'_{js} \equiv \frac{\partial a_s^{[l]}}{\partial x_j}^{(i)} \quad \text{and} \quad z'_{js} \equiv \frac{\partial z_s^{[l]}}{\partial x_j}^{(i)} \quad \forall \quad s \in [1, n^{[l-1]}] \tag{5}$$

In turn, vector quantities will be defined in **bold** font. For instance, the input and output layers associated with example (i) would be given by:

$$\boldsymbol{a}^{[1](i)} = \boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_{n_x}^{(i)} \end{bmatrix} \qquad \boldsymbol{y}^{(i)} = \begin{bmatrix} y_1^{(i)} \\ \vdots \\ y_{n_y}^{(i)} \end{bmatrix} = \boldsymbol{a}^{[L](i)} \tag{6}$$

3.2 Normalize Training Data

Normalizing the training data can greatly improve performance when either inputs or outputs differ by orders of magnitude. A simple way to improve numerical conditioning is to subtract the mean and divide by the standard deviation of the data:

$$\boldsymbol{\mu}_x = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}^{(i)} \in \mathbb{R}^{n_x} \qquad \boldsymbol{\sigma}_x = \sqrt{\frac{1}{m}\sum_{i=1}^{m} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_x\right)^2} \in \mathbb{R}^{n_x} \tag{7}$$

$$\boldsymbol{\mu}_y = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{y}^{(i)} \in \mathbb{R}^{n_y} \qquad \boldsymbol{\sigma}_y = \sqrt{\frac{1}{m}\sum_{i=1}^{m} \left(\boldsymbol{y}^{(i)} - \boldsymbol{\mu}_y\right)^2} \in \mathbb{R}^{n_y} \tag{8}$$

The normalized data then becomes:

$$\boldsymbol{x}^{(i)} := \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_x}{\boldsymbol{\sigma}_x} \in \mathbb{R}^{n_x} \qquad \boldsymbol{y}^{(i)} := \frac{\boldsymbol{y}^{(i)} - \boldsymbol{\mu}_y}{\boldsymbol{\sigma}_y} \in \mathbb{R}^{n_y} \qquad y'^{(i)}_{jk} := \frac{\partial y_k}{\partial x_j}^{(i)} \times \frac{\sigma_{x_j}}{\sigma_{y_k}} \in \mathbb{R} \tag{9}$$

This approach works well, except when all values are very close such that the denominator vanishes as standard deviation goes to zero. In that case, alternative normalization methods (or none) should be considered.

3.3 Forward Propagation

Predictions are obtained by propagating information forward throughout the neural network in a recursive manner, starting with the input layer and ending with the output layer. For each layer (excluding the input layer), activations are computed as follows:

$$a_r = g(z_r) \quad \text{where} \quad z_r = w_{rs}a_s + b_r \quad \forall\, l \in (1, L] \quad \forall\, r \in [1, n^{[l]}] \quad \forall\, s \in [1, n^{[l-1]}] \tag{10}$$

$$a'_{jr} = g'(z_r)z'_{jr} \quad \text{where} \quad z'_{jr} = w_{rs}a'_{js} \quad \forall\, l \in (1, L] \quad \forall\, r \in [1, n^{[l]}] \quad \forall\, s \in [1, n^{[l-1]}] \quad \forall\, j \in [1, n_x] \tag{11}$$

For the hidden layers ($1 < l < L$), the activation function is taken to be the hyperbolic tangent[2]:

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad \Rightarrow \quad g'(z) \equiv \frac{\partial g}{\partial z} = 1 - g(z)^2 \quad \Rightarrow \quad g''(z) \equiv \frac{\partial^2 g}{\partial z^2} = -2g(z)g(z)' \tag{12}$$

and, for the output layer ($l = L$), the activation function is linear:

$$g(z) = z \quad \Rightarrow \quad g'(z) = 1 \quad \Rightarrow \quad g''(z) = 0 \tag{13}$$

3.4 Parameter Update

The neural network parameters are learned by solving the following, unconstrained optimization problem:

$$\theta = \arg\min_{\theta} \mathcal{J}(\theta) \quad \text{where} \quad \theta = \{w_{rs}, b_r\} \quad \forall \quad r \in [1, n^{[l]}] \quad \forall \quad l \in (1, L] \tag{14}$$

The cost function $\mathcal{J}$ is given by the following expression, where λ is a hyper-parameter to be tuned which controls regularization:

$$\mathcal{J}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}^{(i)} + \frac{\lambda}{2m}\sum_{l=2}^{L}\sum_{r=1}^{n^{[l]}}\sum_{s=1}^{n^{[l-1]}} w_{rs}^2 \tag{15}$$

The parameters are updated iteratively for each layer using gradient-descent[3]:

$$\theta := \theta - \alpha\frac{\partial \mathcal{J}}{\partial \theta} \tag{16}$$

[2]Other choices are possible, provided the activation function is smooth (*i.e.* not ReLu)

[3]In practice, some improved version of Gradient-Descent (GD) would be used (*e.g.* ADAM [14]) but GD is easier to explain

where the hyper-parameter α is the learning rate. The loss function $\mathcal{L}$ is taken to be a modified version of the Least Squares Estimator (LSE), augmented with an additional term to account for partial derivatives:

$$l = L: \quad \mathcal{L}^{(i)} = \frac{1}{2}\sum_{r=1}^{n_y}\left[\beta_r^{(i)}\left(a_r^{[L](i)} - y_r^{(i)}\right)^2 + \sum_{j=1}^{n_x}\gamma_{rj}^{(i)}\left(a_{jr}^{\prime[L](i)} - y_{jr}^{\prime(i)}\right)^2\right] \tag{17}$$

where two more hyper-parameters have been introduced, $\beta^{(i)}$ and $\gamma^{(i)}$, which can be used to prioritize partials over values and *vice versa*. By default, $\beta_r^{(i)} = \gamma_{rj}^{(i)} = 1$, meaning partials and values are prioritized equally, which is the common use-case. However, there are situations where there may be an incentive to prioritize one term or the other, as will be shown in the results section. This formulation offers the flexibility to do so.

3.5 Backward Propagation

Forward propagation seeks to compute the gradient of the predicted outputs $\hat{\boldsymbol{y}}$ with respect to (abbreviated w.r.t.) the inputs $\boldsymbol{x}$. By contrast, back-propagation computes the gradient of the cost function $\mathcal{J}$ with respect to the neural net parameters θ:

$$\frac{\partial \mathcal{J}}{\partial \theta} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta} + \frac{\lambda}{2m}\sum_{l=2}^{L}\sum_{r=1}^{n^{[l]}}\sum_{s=1}^{n^{[l-1]}}\frac{\partial}{\partial \theta}\left(w_{rs}^2\right) \tag{18}$$

3.5.1 Loss Function Derivatives w.r.t. Neural Net Parameters

In a first step, assume the quantities $\partial\mathcal{L}/\partial a_r$ and $\partial\mathcal{L}/\partial a'_{jr}$ are known and consider the gradient of the loss function $\mathcal{L}$ with respect to the parameters θ from layer l. Applying the chain rule to Eq. 17 yields:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial a_r}\frac{\partial a_r}{\partial z_r}\frac{\partial z_r}{\partial \theta} + \sum_{j=1}^{n_x}\frac{\partial \mathcal{L}}{\partial a'_{jr}}\left(\frac{\partial a'_{jr}}{\partial z_r}\frac{\partial z_r}{\partial \theta} + \frac{\partial a'_{jr}}{\partial z'_{jr}}\frac{\partial z'_{jr}}{\partial \theta}\right) \tag{19}$$

where the superscript (i) has been dropped for notational convenience, with the understanding that all equations to follow apply to example (i). The various partial derivatives follow from Eqs. 10–11:

$$\frac{\partial a_r}{\partial z_r} = g'(z_r) \qquad \frac{\partial a'_{jr}}{\partial z_r} = g''(z_r)z'_{jr} \qquad \frac{\partial a'_{jr}}{\partial z'_{jr}} = g'(z_r) \tag{20}$$

$$\theta = w_{rs}: \quad \frac{\partial z_r}{\partial \theta} = a_s \qquad \frac{\partial z'_{jr}}{\partial \theta} = a'_{js} \tag{21}$$

$$\theta = b_r: \quad \frac{\partial z_r}{\partial \theta} = 1 \qquad \frac{\partial z'_{jr}}{\partial \theta} = 0 \tag{22}$$

Substituting the above in equation 19 yields:

$$\theta = w_{rs}: \quad \frac{\partial \mathcal{L}}{\partial w_{rs}} = \frac{\partial \mathcal{L}}{\partial a_r}g'(z_r)a_s + \sum_{j=1}^{n_x}\frac{\partial \mathcal{L}}{\partial a'_{jr}}\left(g''(z_r)z'_{jr}a_s + g'(z_r)a'_{js}\right) \tag{23}$$

$$\theta = b_r: \quad \frac{\partial \mathcal{L}}{\partial b_r} = \frac{\partial \mathcal{L}}{\partial a_r}g'(z_r) + \sum_{j=1}^{n_x}\frac{\partial \mathcal{L}}{\partial a'_{jr}}g''(z_r)z'_{jr} \tag{24}$$

3.5.2 Loss Function Derivatives w.r.t. to Neural Net Activations

All quantities in Eqs. 23–24 are known, except for $\partial\mathcal{L}/\partial a_r$ and $\partial\mathcal{L}/\partial a'_{jr}$ which must be obtained recursively by propagating information backwards from one layer to the next, starting with the output layer L. The process is initialized from Eq. 17, where it follows from calculus that:

$$l = L: \quad \frac{\partial \mathcal{L}}{\partial a_r} = \beta_r\left(a_r - y_r\right) \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial a'_{jr}} = \gamma_{rj}\left(a'_{jr} - y'_r\right) \tag{25}$$

The derivatives for the next layer are obtained through the chain rule. According to Eq. 17, the loss function $\mathcal{L}$ depends on $n^{[l]}$ activations a_r and $n_x \times n^{[l]}$ associated partials a'_{jr}, which are themselves functions of a_s and a'_{js}. Hence, according to the chain rule, current layer derivatives w.r.t. to previous layer activations are:

$$\frac{\partial \mathcal{L}}{\partial a_s}^{[l]} = \sum_{r=1}^{n^{[l]}} \left[\frac{\partial \mathcal{L}}{\partial a_r} \frac{\partial a_r}{\partial a_s} + \sum_{j=1}^{n_x} \frac{\partial \mathcal{L}}{\partial a'_{js}} \frac{\partial a'_{js}}{\partial a_s} \right] \quad \forall\ l = L, \dots, 2 \tag{26}$$

$$\frac{\partial \mathcal{L}}{\partial a'_{js}}^{[l]} = \sum_{r=1}^{n^{[l]}} \left[\frac{\partial \mathcal{L}}{\partial a_r} \frac{\partial a_r}{\partial a'_{js}} + \sum_{j=1}^{n_x} \frac{\partial \mathcal{L}}{\partial a'_{jr}} \frac{\partial a'_{jr}}{\partial a'_{js}} \right] \quad \forall\ l = L, \dots, 2 \tag{27}$$

Further applying the chain rule to the intermediate quantities and using Eqs. 10–11, one finds that:

$$\frac{\partial a_r}{\partial a_s} = \frac{\partial a_r}{\partial z_r} \frac{\partial z_r}{\partial a_s} + \frac{\partial a_r}{\partial z'_{jr}} \cancelto{0}{\frac{\partial z'_{jr}}{\partial a_s}} = g'(z_r) w_{rs} \tag{28}$$

$$\frac{\partial a_r}{\partial a'_{js}} = \cancelto{0}{\frac{\partial a_r}{\partial z_r}} \frac{\partial z_r}{a'_{js}} + \cancelto{0}{\frac{\partial a_r}{\partial z'_{jr}}} \frac{\partial z'_{jr}}{a'_{js}} = 0 \tag{29}$$

$$\frac{\partial a'_{jr}}{\partial a_s} = \frac{\partial a'_{jr}}{\partial z_r} \frac{\partial z_r}{\partial a_s} + \frac{\partial a'_{jr}}{\partial z'_{jr}} \cancelto{0}{\frac{\partial z'_{jr}}{\partial a_s}} = g''(z_r) z'_{jr} w_{rs} \tag{30}$$

$$\frac{\partial a'_{jr}}{\partial a'_{js}} = \frac{\partial a'_{jr}}{\partial z_r} \cancelto{0}{\frac{\partial z_r}{\partial a'_{js}}} + \frac{\partial a'_{jr}}{\partial z'_{jr}} \frac{\partial z'_{jr}}{\partial a'_{js}} = g'(z_r) w_{rs} \tag{31}$$

All necessary equations to compute the derivatives of the loss function w.r.t. parameters in any layer are now known. Derivatives are obtained by working our way back recursively across layers, repeatedly applying Eqs. 26–27 after passing the solution in the current layer to the preceding layer as follows:

$$\frac{\partial \mathcal{L}}{\partial a_r}^{[l-1]} = \frac{\partial \mathcal{L}}{\partial a_s}^{[l]} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial a'_{jr}}^{[l-1]} = \frac{\partial \mathcal{L}}{\partial a'_{js}}^{[l]} \quad \forall\ l = L, \dots, 2 \tag{32}$$

3.5.3 Cost Function Derivatives w.r.t. Parameters

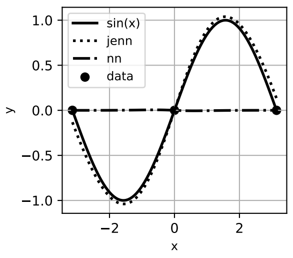
Putting it all together, the derivative of the cost function w.r.t. neural parameters are obtained by substituting Eqs. 23–24 into Eq. 18, where the subscripts r and s are referenced to layer $[l]$:

$$\frac{\partial \mathcal{J}}{\partial w_{rs}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}^{(i)}}{\partial w_{rs}} + \lambda w_{rs} \qquad \text{and} \qquad \frac{\partial \mathcal{J}}{\partial b_r} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}^{(i)}}{\partial b_r} \qquad \forall\ l \in (1, L] \tag{33}$$

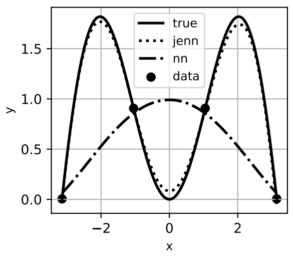
4 Verification and Validation

The theory developed in the previous section was implemented into an installable Python library for which a link was provided earlier. This section shows validation results[4] for selected test functions in Fig. 2, which compares JENN against a standard Neural Net (NN) when training data is sparse (shown as dots). Higher dimensional examples are available on the project's homepage, but were not included since it is hard to visualize more than two dimensions on a static plot. As can be seen, all else being equal except for jacobian-enhancement, JENN consistently outperforms NN in all cases. Finally, it was also verified that runtime scales as $\mathcal{O}(n)$ using the Rastrigin function to generate progressively larger datasets and training a new model, as shown in Fig. 3 using a last generation MacBook Pro.

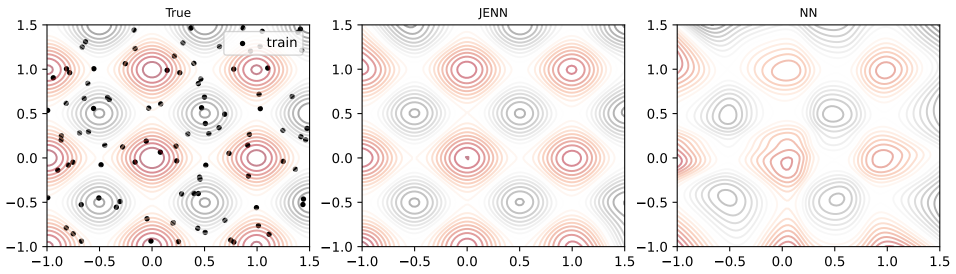
[4]The code used to generate these results can be found at `https://github.com/shb84/JENN/tree/master/docs/examples`



(a) $y = sin(x)$



(b) $y = xsin(x)$



(c) $y = \sum_{i=1}^{2} \left[x_i^2 - 10 \cos(2\pi x_i) + 10 \right]$

Figure 2: Validation Results

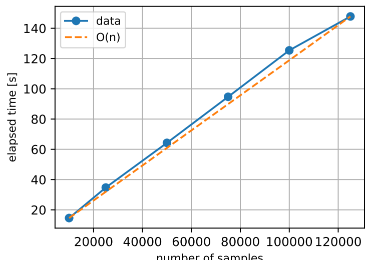


Figure 3: Runtime *vs.* sample size

5 Example Application: Surrogate-Based Optimization

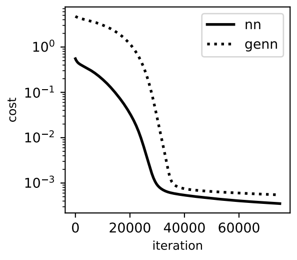
Surrogate-based optimization is the process of using a surrogate model to optimize some function of interest, rather than the function itself. The so-called surrogate model is a fast-running approximation of the true response, such as a neural net. This use-case typically arises when the response of interest takes too long to evaluate relative to the needs of the application. For example, in real-time optimal control, one may wish to consider a neural network to make quick predictions of complicated physics when updating the trajectory. The purpose of this section is to quantify the benefit of JENN relative to regular neural networks for gradient-based optimization. This will be accomplished by solving the following optimization problem:

$$\begin{aligned} \text{Minimize:} \quad & \hat{y} = f(x_1, x_2) && (34) \\ \text{With Respect To:} \quad & x_1 \in \mathbb{R}, x_2 \in \mathbb{R} && (35) \\ \text{Subject To:} \quad & -2 \leq x_1 \leq 2 && (36) \\ & -2 \leq x_2 \leq 2 && (37) \end{aligned}$$

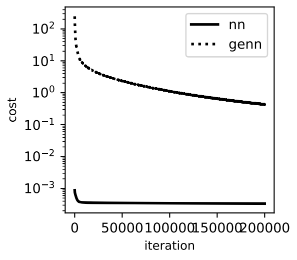
where $\hat{y}$ is a surrogate-model approximation to the famous Rosenbrock function:

$$y = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \tag{38}$$

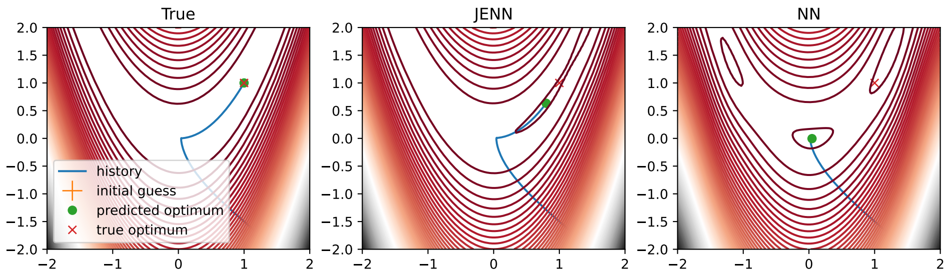
Results[5] are shown in Fig.4 where optimizer convergence histories are shown for the true response and its surrogate-model approximations using JENN and NN under the same settings, except for jacobian-enhancement, respectively. Training data (not shown to avoid clutter) was generate by sampling the true response on a 9×9 regular grid. Training histories are provided in Figs. 4a–4b to verify that both neural networks converged to the same degree.



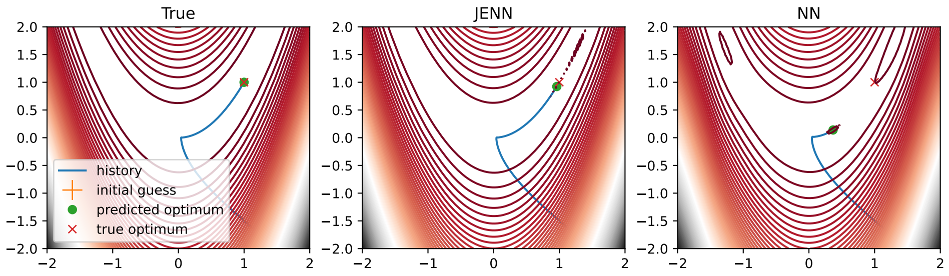
(a) Training history (without polishing)



(b) Training history (with polishing)



(c) Surrogate-based optimization (without polishing)



(d) Surrogate-based optimization (with polishing)

Figure 4: Caption place holder

5.1 Results Without Polishing

In a first step, consider Fig.4c which does not use polishing (explained in the next section). Under perfect conditions, one would expect the optimization history to follow the same path for all models, but this is not observed. For both surrogates, the terminal value is indeed close to zero (the true optimum) but neither JENN nor NN reach the true location of this optimum at $(1, 1)$; although JENN gets much closer, as expected. The Rosenbrock function is notorious for exhibiting a shallow valley with near-zero slope, which creates a significant challenge for surrogate-based optimization. One can indeed observe slight undulations in that valley, within the noise of surrogate prediction error, which create artificial local minima that trap the optimizer. From Eq. 17, recall that neural nets learn by minimizing prediction error. As a result, by construction, there is more reward during training for accurately predicting regions of large change than

[5]The code used to generate these results can be found at `https://github.com/shb84/JENN/tree/master/docs/examples`

there is for accurately predicting small features in the vicinity of the optimum. This situation is at odds with the needs of gradient-based optimization, which requires extreme accuracy near the optimum; accuracy elsewhere is desirable but less critical, provided slopes correctly point in the direction of improvement.

5.2 Results With Polishing

For most practical engineering problems, close is good enough. However, there are situations where optimization accuracy is desirable if not critical, such as airfoil design (to ensure smooth contours free of undulations). This situation can be ameliorated by using what will be called polishing. Polishing is the process of restarting a previously trained model and continue training it after magnifying portions of the design space, where either function values or partials are very small. This can be accomplished by judiciously setting 'beta' or 'gamma' to prioritize specific training points. For example, a radial basis function could be used:

$$\beta^{(i)} = 1 + 1000 \times e^{-\left(\epsilon y_k^{(i)}\right)^2} \quad \forall \quad k \in [1, n_y],\ i \in [1, m] \tag{39}$$

$$\gamma^{(i)} = 1 + 1000 \times e^{-\left(\epsilon \frac{\partial y_k}{\partial x_j}^{(i)}\right)^2} \quad \forall \quad j \in [1, n_x], k \in [1, n_y],\ i \in [1, m] \tag{40}$$

where ϵ is another hyperparameter, set to $\epsilon = 0.1$ in the results shown, which control the radius of influence of a point. Far away from any point $x^{(i)}$ where $y^{(i)} = \partial y / \partial x^{(i)} = 0$, the weights are $\gamma^{(i)} = \beta^{(i)} \approx 1$, whereas nears those points $\gamma^{(i)} = \beta^{(i)} \approx 1001$. This has the desired effect of disproportionately emphasizing small values of y_k and $\frac{\partial y_k}{\partial x_j}$ during training (by up to a selected factor of 1000). The results are shown in Fig. 4d where it can be seen that JENN now recovers the true optimize almost exactly; artificial local minima nearly vanish. This is not true for NN; although results improve noticeably. Overall, this section demonstrates that jacobian-enhancement offer a quantifiable benefit over standard neural nets for SBO applications.

6 Conclusions

In summary, this research developed a deep-learning formulation for jacobian-enhanced multi-layer perceptrons, exclusively composed of dense layers, and successfully validated the implementation against selected canonical test functions. It was shown that validation results support the general consensus that gradient-enhanced methods require less training data and achieve better accuracy than their non-gradient-enhanced counterparts, provided the cost of obtaining training data partials is not prohibitive in the first place. However, in the context of surrogate-based optimization, it was observed that despite significant improvement over standard neural nets, a learning procedure which minimizes some version of least squares estimator is at odds with the needs of gradient descent in the vicinity of the optimum, where slopes a close to zero. It was shown that this shortcoming can be overcome through a process termed *polishing*, which prioritizes individual sample points with near-zero slopes through judicious use of hyperparameters. This unique feature enabled by the present formulation thus provides an additional *knob* for tuning accuracy in situations that call for it. The associated, open-source Python library was provided as a link in earlier sections in the hope that it will be useful to others and spark new contributions in this researcher area.

Acknowledgement

The supporting code for this work used the exercises by Prof. Andrew Ng in the *Coursera Deep Learning Specialization*[6] as a starting point. It then built upon it to include additional features but, most of all, it fundamentally changed the formulation to include gradient-enhancement and made sure all arrays were updated in place (data is never copied). The author would therefore like to thank Andrew Ng for offering the fundamentals of deep learning on Coursera, which took a complicated subject and explained it in simple terms that even an aerospace engineer could understand.

[6]`https://www.coursera.org/specializations/deep-learning`

A Key Equations in Vectorized Form

Key equations from the preceding material will now be re-written in a vectorized form (*i.e.* processing all m examples simultaneously without loops) suitable for programming, as a helpful companion to source code. In the notation to follow, the symbol $\odot$ implies element-wise array multiplication.

A.1 Data Structures

Using capital letters, notation can be extended to matrix notation to account for all m training examples, starting with training data inputs and outputs:

$$\boldsymbol{X} = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_{n_x}^{(1)} & \dots & x_{n_x}^{(m)} \end{bmatrix} \in \mathbb{R}^{n_x \times m} \qquad \boldsymbol{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_1^{(m)} \\ \vdots & \ddots & \vdots \\ y_{n_y}^{(1)} & \dots & y_{n_y}^{(m)} \end{bmatrix} \in \mathbb{R}^{n_y \times m} \tag{41}$$

The associated Jacobians are formatted as:

$$\boldsymbol{J} = \begin{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1}^{(1)} & \dots & \frac{\partial y_1}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial y_1}{\partial x_{n_x}}^{(m)} \end{bmatrix} & \dots & \begin{bmatrix} \frac{\partial y_{n_y}}{\partial x_1}^{(1)} & \dots & \frac{\partial y_{n_y}}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{n_y}}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial y_{n_y}}{\partial x_{n_x}}^{(m)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{n_y \times n_x \times m} \tag{42}$$

The shape of the neural network parameters to be learned (*i.e.* weights and biases) is independent of the number of training examples. They depend only on layer sizes and are denoted as:

$$\mathbf{W}^{[l]} = \begin{bmatrix} w_{11}^{[l]} & \dots & w_{1n^{[l-1]}}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{n^{[l]}1}^{[l]} & \dots & w_{n^{[l]}n^{[l-1]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}} \qquad \boldsymbol{b}^{[l]} = \begin{bmatrix} b_1^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times 1} \quad \forall \quad l \in (1, L] \tag{43}$$

However, hidden layer activations do depend on the number of examples and are stored as:

$$\mathbf{A}^{[l]} = \begin{bmatrix} a_1^{(1)} & \dots & a_1^{(m)} \\ \vdots & \ddots & \vdots \\ a_{n^{[l]}}^{(1)} & \dots & a_{n^{[l]}}^{(m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m} \qquad \mathbf{Z}^{[l]} = \begin{bmatrix} z_1^{(1)} & \dots & z_1^{(m)} \\ \vdots & \ddots & \vdots \\ z_{n^{[l]}}^{(1)} & \dots & z_{n^{[l]}}^{(m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times m} \tag{44}$$

The associated hidden layer derivatives w.r.t. inputs are stored as:

$$\mathbf{A}'^{[l]} = \begin{bmatrix} \begin{bmatrix} \frac{\partial a_1}{\partial x_1}^{(1)} & \dots & \frac{\partial a_1}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_1}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial a_1}{\partial x_{n_x}}^{(m)} \end{bmatrix} & \dots & \begin{bmatrix} \frac{\partial a_{n^{[l]}}}{\partial x_1}^{(1)} & \dots & \frac{\partial a_{n^{[l]}}}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_{n^{[l]}}}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial a_{n^{[l]}}}{\partial x_{n_x}}^{(m)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times n_x \times m} \tag{45}$$

$$\mathbf{Z}'^{[l]} = \begin{bmatrix} \begin{bmatrix} \frac{\partial z_1}{\partial x_1}^{(1)} & \dots & \frac{\partial z_1}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial z_1}{\partial x_{n_x}}^{(m)} \end{bmatrix} & \dots & \begin{bmatrix} \frac{\partial z_{n^{[l]}}}{\partial x_1}^{(1)} & \dots & \frac{\partial z_{n^{[l]}}}{\partial x_1}^{(m)} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_{n^{[l]}}}{\partial x_{n_x}}^{(1)} & \dots & \frac{\partial z_{n^{[l]}}}{\partial x_{n_x}}^{(m)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{n^{[l]} \times n_x \times m} \tag{46}$$

It follows that hidden layer derivatives w.r.t. a specific input x_j are formatted as:

$$\mathbf{A}_j'^{[l]} = \begin{bmatrix} \dfrac{\partial a_1}{\partial x_j}^{(1)} & \cdots & \dfrac{\partial a_1}{\partial x_j}^{(m)} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial a_{n^{[l]}}}{\partial x_j}^{(1)} & \cdots & \dfrac{\partial a_{n^{[l]}}}{\partial x_j}^{(m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]}\times m} \qquad \mathbf{Z}_j'^{[l]} = \begin{bmatrix} \dfrac{\partial z_1}{\partial x_j}^{(1)} & \cdots & \dfrac{\partial z_1}{\partial x_j}^{(m)} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial z_{n^{[l]}}}{\partial x_j}^{(1)} & \cdots & \dfrac{\partial z_{n^{[l]}}}{\partial x_j}^{(m)} \end{bmatrix} \in \mathbb{R}^{n^{[l]}\times m} \tag{47}$$

Finally, the back propagation derivatives w.r.t. parameters are given by:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{[l]}} = \begin{bmatrix} \dfrac{\partial \mathcal{J}}{\partial w_{11}} & \cdots & \dfrac{\partial \mathcal{J}}{\partial w_{1n^{[l-1]}}} \\ \vdots & \vdots & \vdots \\ \dfrac{\partial \mathcal{J}}{\partial w_{n^{[l]}1}} & \cdots & \dfrac{\partial \mathcal{J}}{\partial w_{n^{[l]}n^{[l-1]}}} \end{bmatrix} \in \mathbb{R}^{n^{[l]}\times n^{[l-1]}} \quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{b}^{[l]}} = \begin{bmatrix} \dfrac{\partial \mathcal{J}}{\partial b_1} \\ \vdots \\ \dfrac{\partial \mathcal{J}}{\partial b_{n^{[l]}}} \end{bmatrix} \in \mathbb{R}^{n^{[l]}\times 1} \quad \forall \quad l \in (1, L] \tag{48}$$

Also, the following hyperparameters can be either scalars or arrays, depicted here as the latter:

$$\beta = \begin{bmatrix} \beta_1^{(1)} & \cdots & \beta_1^{(m)} \\ \vdots & \vdots & \vdots \\ \beta_{n_y}^{(1)} & \cdots & \beta_{n_y}^{(m)} \end{bmatrix} \in \mathbb{R}^{n_y\times m} \quad \gamma = \begin{bmatrix} \begin{bmatrix} \gamma_{11}^{(1)} & \cdots & \gamma_{11}^{(m)} \\ \vdots & \ddots & \vdots \\ \gamma_{1n_x}^{(1)} & \cdots & \gamma_{1n_x}^{(m)} \end{bmatrix} & \cdots & \begin{bmatrix} \gamma_{n_y1}^{(1)} & \cdots & \gamma_{n_y1}^{(m)} \\ \vdots & \ddots & \vdots \\ \gamma_{n_yn_x}^{(1)} & \cdots & \gamma_{n_yn_x}^{(m)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{n_y\times n_x\times m} \tag{49}$$

A.2 Forward Propagation (Eqs. 10–11)

$$\mathbf{A}^{[l]} = g\left(\mathbf{Z}^{[l]}\right) \qquad \text{where} \quad \mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \odot \mathbf{A}^{[l-1]} + \boldsymbol{b}^{[l]} \quad \forall \quad l \in (1, L] \tag{50}$$

$$\mathbf{A}_j^{'[l]} = g\left(\mathbf{Z}^{[l]}\right) \odot \mathbf{Z}_j^{'[l]} \ \text{where} \quad \mathbf{Z}^{'[l]} = \mathbf{W}^{[l]} \odot \mathbf{A}_j^{'[l-1]} + \boldsymbol{b}^{[l]} \quad \forall \quad j \in [1, n_x] \tag{51}$$

A.3 Parameter Update (Eq. 16)

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{[l]}} \quad \text{and} \quad \boldsymbol{b}^{[l]} := \boldsymbol{b}^{[l]} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{b}^{[l]}} \quad \forall \ l \in (1, L] \quad \forall \ \alpha \in \mathbb{R} \tag{52}$$

A.4 Backward Propagation (Eqs. 23–25, 26–27, 33)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[L]}} = \beta \odot \left(\mathbf{A}^{[L]} - \mathbf{Y}\right) \tag{53}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}'^{[L]}} = \gamma \odot \left(\mathbf{A}'^{[L]} - \mathbf{Y}'\right) \tag{54}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l-1]}} = \mathbf{W}^{\top[l]} \cdot \left(g'\left(\mathbf{Z}^{[l]}\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l]}}\right) + \sum_{j=1}^{n_x} \mathbf{W}^{\top[l]} \cdot \left(g''\left(\mathbf{Z}^{[l]}\right) \odot \mathbf{Z}_j'^{[l]} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{A}_j'^{[l]}}\right) \tag{55}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A_j}'^{[l-1]}} = \sum_{j=1}^{n_x} \mathbf{W}^{\top[l]} \cdot \left(g'\left(\mathbf{Z}^{[l]}\right) \odot \frac{\partial \mathcal{L}}{\partial \mathbf{A}_j'^{[l]}}\right) \tag{56}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l]}} \odot g'\left(\mathbf{Z}^{[l]}\right) \odot \mathbf{A}^{[l-1]} + \sum_{j=1}^{n_x} \frac{\partial \mathcal{L}}{\partial \mathbf{A}_j'^{[l]}} \odot \left[g''\left(\mathbf{Z}^{[l]}\right) \odot \mathbf{Z}_j'^{[l]} \odot \mathbf{A}'^{[l-1]} + g'\left(\mathbf{Z}^{[l]}\right) \mathbf{A}_j'^{[l-1]}\right] \tag{57}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l]}} \odot g'\left(\mathbf{Z}^{[l]}\right) + \sum_{j=1}^{n_x} \frac{\partial \mathcal{L}}{\partial \mathbf{A}_j'^{[l]}} \odot g''\left(\mathbf{Z}^{[l]}\right) \odot \mathbf{Z}_j'^{[l]} \tag{58}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} S\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}\right) + \frac{\lambda}{m} \mathbf{W}^{[l]} \tag{59}$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}^{[l]}} = \frac{1}{m} S\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}\right) \tag{60}$$

where S is used to denote the vectorized summation along the last axis of an array[7]. Any other summation shown implies that it could not be vectorized and required a loop in the code. This only happened when summing over partials. Hence, the number of loops needed equals the number of inputs, which is always negligible compared to the size of the training data m. The latter is the critical dimension to be vectorized.

References

[1] Keane, A. J. and Nair, P. B., "Computational Approaches for Aerospace Design Computational," chap. 5, 7, Wiley, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, 2005, pp. 211–270, 301–326.

[2] Forrester, A. I. J., Sobester, A., and Keane, A. J., *Engineering Design via Surrogate Modelling - A Practical Guide.*, Wiley, 2008.

[3] Giannakoglou, K., Papadimitriou, D., and Kampolis, I., "Aerodynamic shape design using evolutionary algorithms and new gradient-assisted metamodels," *Computer Methods in Applied Mechanics and Engineering*, Vol. 195, No. 44, 2006, pp. 6312–6329.

[4] Bouhlel, M. A., He, S., and Martins, J. R. R. A., "Scalable gradient–enhanced artificial neural networks for airfoil shape design in the subsonic and transonic regimes," *Structural and Multidisciplinary Optimization*, Vol. 61, No. 4, 2020, pp. 1363–1376.

[5] Nagawkar, J. R., Leifsson, L. T., and He, P., *Aerodynamic Shape Optimization Using Gradient-Enhanced Multifidelity Neural Networks.*

[6] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, Software available from tensorflow.org.

[7]`https://numpy.org/doc/stable/reference/generated/numpy.sum.html`

[7] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, Vol. 12, 2011, pp. 2825–2830.

[8] Saves, P., Lafage, R., Bartoli, N., Diouane, Y., Bussemaker, J., Lefebvre, T., Hwang, J. T., Morlier, J., and Martins, J. R. R. A., "SMT 2.0: A Surrogate Modeling Toolbox with a focus on Hierarchical and Mixed Variables Gaussian Processes," *Advances in Engineering Sofware*, Vol. 188, 2024, pp. 103571.

[9]

[10] Laurent, L., Le Riche, R., Soulier, B., and Boucard, P.-A., "An Overview of Gradient-Enhanced Metamodels with Applications," *Archives of Computational Methods in Engineering*, Vol. 26, No. 1, 2019, pp. 61–106.

[11] Sellar, R. and Batill, S., *Concurrent Subspace Optimization using gradient-enhanced neural network approximations*.

[12] Liu, W. and Batill, S., *Gradient-enhanced neural network response surface approximations*.

[13] Yu, J., Lu, L., Meng, X., and Karniadakis, G. E., "Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems," *Computer Methods in Applied Mechanics and Engineering*, Vol. 393, 2022, pp. 114823.

[14] Kingma, D. P. and Ba, J., "Adam: A Method for Stochastic Optimization," 2017.