2022 IEEE/SICE International Symposium on System Integration (SII) | 978-1-6654-4540-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SII52469.2022.9708840

A Symmetric Neural Network to Compute Fractional Derivatives by Training with Integer Derivatives

Tan Chen[1] and Bill Goodwine[2]

Abstract— **Fractional calculus is an increasingly recognized important tool for modeling complicated dynamics in modern engineering systems. While, in some ways, fractional derivatives are a straight-forward generalization of integer-order derivatives that are ubiquitous in engineering modeling, in other ways the use of them requires quite a bit of mathematical expertise and familiarity with some mathematical concepts that are not in everyday use across the broad spectrum of engineering disciplines. In more colloquial terms, the learning curve is steep. While the authors recognize the need for fundamental competence in tools used in engineering, a computational tool that can provide an alternative means to compute fractional derivatives does have a useful role in engineering modeling. This paper presents the use of a symmetric neural network that is trained entirely on integer-order derivatives to provide a means to compute fractional derivatives. The training data does not contain any fractional-order derivatives at all, and is composed of only integer-order derivatives. The means by which a fractional derivative can be obtained is by requiring the neural network to be symmetric, that is, it is the composition of two identical sets of layers trained on integer-order derivatives. From that, the information contained in the nodes between the two sets of layers contains half-order derivative information.**

I. INTRODUCTION

Fractional calculus is increasingly being used to model modern engineering systems and the literature is fairly extensive and only an overview can be provided here. There are several books overviewing the topic from a mathematical perspective, including [10], [9], [11]. Some papers considering large-scale and infinite order dynamics include, for example, [4], [12], [8]. Modeling viscoelastic systems is also an obvious application [2], [5]. The authors have used it to model very large scale robotic systems [3], [6], [7]. Fractional-order control is also a topical area such as in [14], [1]. An excellent review article illustrating the very broad range of applications of fractional calculus and control in science and engineering is [13]. In Section II we provide an overview of the basic concepts. While these are fairly straightforward, for an engineer first dealing with a fractional-order system, the learning curve is a bit steep and may prevent immediate access to the utility of fractional-order modeling. In this paper, we present a tool meant in practice as a sort of segue for the uninitiated, which is a neural network trained on data that is made up entirely of integer-order derivatives, *i.e.*, the neural network is *not* trained on fractional-order derivatives, but provides fractional-derivative information.

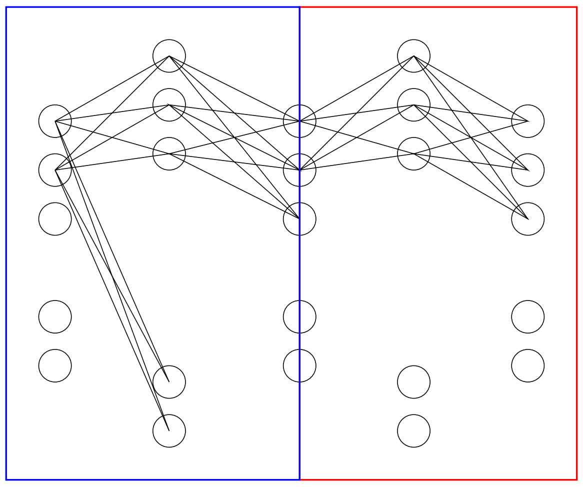

Fig. 1. A neural network that is symmetric if the parts in the red and blue boxes are identical.

Fractional derivatives are generalizations of integer-order derivatives, and, as is the case with generalizations, there is not necessarily a unique one. The use of fractional calculus in engineering and applied science is such that there is not yet a clear consensus as to which derivative is "best" in the sense of being the most useful. In fact, as of the date of this article, the English Wikipedia Fractional Calculus page lists approximately 20 different fractional derivative definitions. In this paper we will take a simple approach in that a half derivative is an operation that, if applied twice, will result in the usual first derivative of a function. We will also focus, for this preliminary study, on a class of functions containing polynomials only, for which many of the various types of definitions of fractional derivatives yield identical results.

Our approach is fairly simple. In the usual case for a neural network, the relationship between a set of input nodes and output nodes is "learned" by determining a set of weights for edges connecting the nodes, including possibly some "hidden" or intermediate layers of nodes, that best fit some training data. So, the usual scenario is illustrated in Figure 1. For example, if we wanted to use the network to compute an approximate derivative, we train the network on many pairs of functions and their derivatives, and the network essentially learns the derivative operator.

If we want to compute a half derivative, then one option would be to train the network on a data set of functions and

*The support of the US National Science Foundation under grant No. CMMI 1826079 is gratefully acknowledged.

[1]Tan Chen is with the Coordinated Science Laboratory, University of Illinois Urbana-Champaign, Urbana, IL USA tanchen@illinois.edu

[2]Bill Goodwine is with the Department of Aerospace and Mechanical Engineering, University of Notre Dame, Notre Dame, IN USA goodwine@controls.ame.nd.edu

their half derivatives. That would be potentially useful to use the network to compute half derivatives of some functions for which the user does not have half derivative information. A more useful scenario is, though, that the user does not have access to *any* half derivative information. In that case, if we place two identical networks in series and train the combination on first-order derivatives, then, by definition, what one of the two networks making up the system would compute is the operation that, if done twice, is the first derivative, *i.e.*, it is the half derivative. In Figure 1, this would require that the networks contained in the left and right boxes (blue and red, respectively), be identical.

The next section outlines some of the basics of fractional calculus and feed-forward neural networks. Furthermore, it describes in detail the tools we used to construct and train our neural network. Section III presents the initial results which exhibit undesirable attributes due to the non-uniqueness of the half derivative. Section IV presents the results of training with a merit function that includes a term to "regularize" the middle layer, which yields fairly satisfactory results. Finally, Section V presents our conclusions and outlines possible avenues of future work.

II. Brief Fractional Calculus and Neural Network Background

This section presents the necessary background on fractional calculus needed to interpret the results of this work, the details of the neural network we implemented and the tools used to do so.

A. *Fractional Calculus*

The basic idea of fractional calculus is simple: what are the operators "in between" integer-order derivatives? The prototypical example would be the half derivative. In the case of polynomials, determining a candidate half derivative is straight-forward.

Consider the monomial

$$f(t) = t^n$$

with the usual sequence of (integer-order) derivatives

$$\begin{aligned} \frac{df}{dt}(t) &= nt^{n-1} \\ \frac{d^2f}{dt^2}(t) &= n(n-1)t^{n-2} \\ &\vdots \\ \frac{d^kf}{dt^k}(t) &= \frac{n!}{(n-k)!}t^{n-k}. \end{aligned} \tag{1}$$

The exponent on the t can take on fractional values, so the hindrance to generalizing this to fractional values of k is the factorial function. Fortunately, as is well-known, that is easily generalized to non-integer values with the gamma function,

$$\Gamma(k) = \int_0^\infty x^{k-1}e^{-x}dx$$

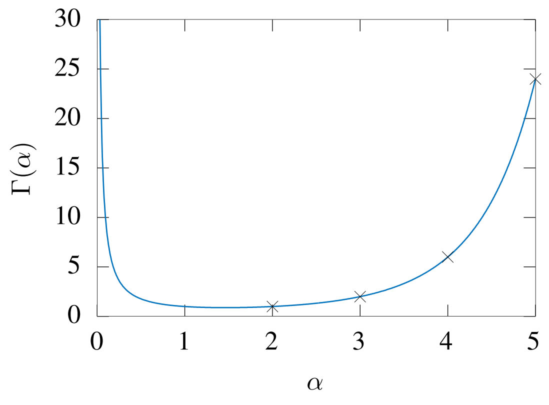

Fig. 2. Gamma function as generalization of the factorial.

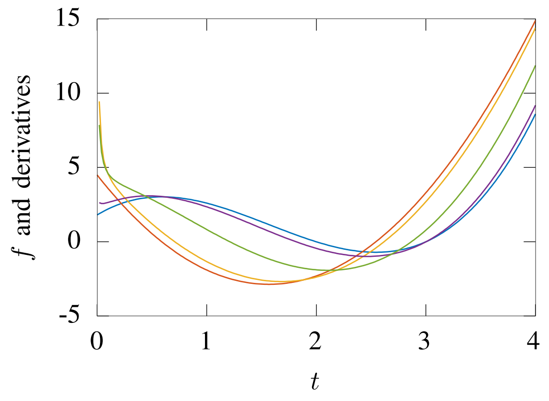

Fig. 3. Example fractional-order derivatives using Equation 2.

which is illustrated in Figure 2, from which it is clear that for integer values

$$k! = \Gamma(k+1),$$

that is, the gamma function generalizes the factorial, but is shifted by one.

Hence, we can replace the factorials in the pattern for the derivatives in Equation 1 with the gamma function

$$\frac{d^\alpha f}{dt^\alpha}(t) = \frac{\Gamma(n+1)}{\Gamma(n-\alpha+1)}t^{n-\alpha}, \tag{2}$$

where $0 < \alpha < n$. We will use this definition to check the results from our symmetric neural network. Figure 3 illustrates $f(t) = t^3 - 4.7t^2 + 4.5t + 1.8$ and its 0.1, 0.5, 0.9 and 1st derivatives. As expected the fractional orders near 0 and 1 are close to the corresponding integer-order values, and the half derivative is "in between" in a very intuitively expected manner.

Unfortunately, beyond simple monomials, computing half derivatives becomes increasingly complicated. The three most common fractional derivatives are as follows. In each case, the notation $\lceil \alpha \rceil$ is notation for the ceiling operator which gives the smallest integer greater than α.

1) The *Riemann-Liouville Fractional Derivative* is given

by

$$ {}^{RL}_{\;0}\mathrm{D}_t^{\alpha} f(t) = \frac{\mathrm{d}^{\lceil\alpha\rceil}}{\mathrm{d}t^{\lceil\alpha\rceil}}\left(\frac{1}{\Gamma\left(\lceil\alpha\rceil-\alpha\right)} \times \int_0^t (t-z)^{\lceil\alpha\rceil-\alpha-1} f(z)\,\mathrm{d}z\right). $$

2) The *Caputo Fractional Derivative* is given by

$$ {}^{C}_{0}\mathrm{D}_t^{\alpha} f(t) = \frac{1}{\Gamma\left(\lceil\alpha\rceil-\alpha\right)} \int_0^t (t-z)^{\lceil\alpha\rceil-\alpha-1} \frac{\mathrm{d}^{\lceil\alpha\rceil} f}{\mathrm{d}z^{\lceil\alpha\rceil}}(z)\,\mathrm{d}z. $$

3) The Grünwald-Letnikov fractional derivative is given by

$$ \frac{\mathrm{d}^{\alpha} f}{\mathrm{d}t^{\alpha}}(t) = \lim_{\Delta t\to 0} \frac{\sum_{k=0}^{\infty}(-1)^k \binom{\alpha}{k} f(t-k\Delta t)}{(\Delta t)^{\alpha}}, $$

where the binomial coefficient is generalized to non-integer values by

$$ \left(\begin{array}{c}\alpha\\ \beta\end{array}\right) = \frac{\Gamma(\alpha+1)}{\Gamma(\beta+1)\,\Gamma(\alpha-\beta+1)}. $$

While utilizing these definitions is certainly possible for most working engineers, this work is meant to provide a simpler computation approach when warranted.

B. Feed-forward Neural Network

Neural networks are a very standard tool in modern engineering, and only a brief description is provided here to set the relevant context for the novel adaptation we implemented for our results.

We utilize a standard feed-forward neural network composed of an input layer, a first hidden layer, a middle layer, a second hidden layer and then an output layer. Each layer is composed of a set of nodes, which contain numerical values, and each node may be connected with an edge to nodes in the subsequent layer. The input layer nodes receive their values as inputs. If a node in the subsequent layer is connected to a node in a previous layer, then the values in each of the node in the previous layers is multiplied by the value of the edge connecting the two nodes, and then all those values are summed in the node. Typically, and activation function is applied to this sum, which then is the value for that node. In our network, we either use no activation function (for the input, middle and output layers) or the rectified linear activation function (typically denoted by ReLU), which in our case simply sets a negative value to zero and leaves any positive value unchanged.

A neural network typically has random values assigned to the edges initially, and training is accomplished by taking known input/output pairs, applying the input and computing the output, comparing the computed output to the correct output to determine an error, and then changing the edge weights to reduce the error. By repeatedly applying the input/output pairs and changing the weights to reduce the error, the network can converge to represent the relationship between the input and output.

Our implementation has an input layer with 200 nodes, a hidden layer with 400 nodes, a middle layer with 200 nodes, a second hidden layer with 400 nodes and then an output layer. If we require that the weights for the edges connecting the input to middle layer be identical to the edge weights connecting the corresponding middle layer nodes to the second hidden layer nodes, and similarly for the hidden layer to the middle layer and the second hidden layer to the output layers, then we have an overall neural network that is made up of two networks in series where we require both networks to be identical.

In our implementation, the 200 input nodes received values of the function we want to differentiate evaluated at 200 time steps. In training, the output layer will represent the first derivative at the same 200 time steps. The 200 nodes in the hidden layer should represent the value of the half derivative if we force the two halves of the network to have an identical structure with identical edge weights.

C. Tools

We used the `pytorch` python library to implement our symmetric neural network. It is fairly straight-forward to make a neural network with two identical halves. Specifically,

```
class SymmetricNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(SymmetricNet, self).__init__()
        self.hidden = torch.nn.Linear(D_in, H)
        self.output = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.hidden(x)
        h_relu = h_relu.clamp(min=0)
        h_relu = self.output(h_relu)
        h_middle = h_relu
        h_relu = self.hidden(h_relu)
        h_relu = h_relu.clamp(min=0)
        y_pred = self.output(h_relu)
        return y_pred, h_middle

model = SymmetricNet(200, 400, 200)
```

Remark 1: Note that we only define one hidden and one output layer. In the forward function we use them both twice, which is the mechanism by which we create the network that is the sequential concatenation of two identical neural networks.

The loss function we use is `MSELoss`, which seeks to minimize the mean squared error. Specifically, the loss function is

$$ error = \sum_{n=0}^{200} \left(prediction(n) - exact(n)\right)^2 \qquad (3) $$

where $prediction(n)$ is the value of the nth node of the output layer and $exact(n)$ is the exact value of the first derivative at time $t = n\Delta t$. The optimizer is `torch.nn.SGD`, which is a stochastic gradient descent method. Note that the ubiquitous back-propagation method can not be used, at least

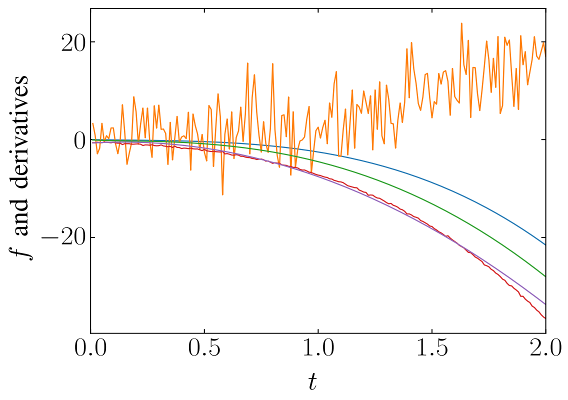

Fig. 4. Initial training results after 2,000 epochs. Blue – input function, orange – predicted half derivative, green – exact half derivative, red – predicted derivative, purple – exact derivative.

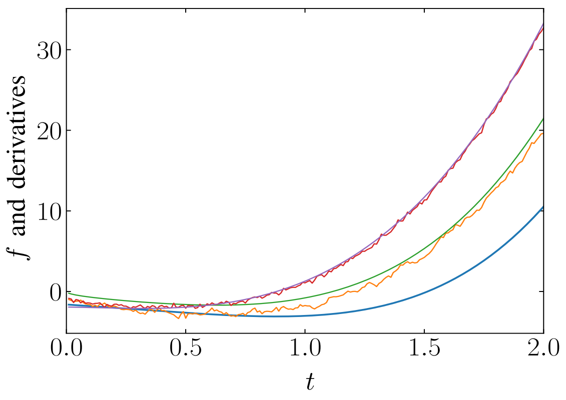

Fig. 5. Typical training result after 1,800 epochs from error function imposing smoothing on middle layer.

directly, because of our constraint that the two halves of our network be identical.[1]

III. INITAL RESULTS

For training data, we generated 10,000 random polynomials. Specifically, the polynomials contained six terms, where each term has a random integer coefficient with values in the range $[-2, 2]$ and random integer powers with values in the range $[0, 4]$. The first-order derivatives were computed and the pairs of polynomials and derivatives were used in training. The half derivative computed by Equation 2 was also computed for validation, but was not used in training. In our training, the 10,000 polynomials are divided into the training dataset (9,000 polynomials) and the validation dataset (1,000 polynomials). In the training process, one *batch* includes the entire training dataset. One *epoch* means that each sample in the training dataset has had an opportunity to update the internal model parameters.

After training so that the mean square error converged to a small value, and validating against a polynomial not used in training, all results were qualitatively of the nature illustrated in Figure 4. The input function is the blue curve. The exact first derivative is the purple curve, and the first derivative computed by the neural network is illustrated by the red curve, which is a close match to the exact first derivative. The half derivative computed using Equation 2 is the green curve, and the output of the middle layer in the orange, "noisy" curve, which is clearly not at all a prediction of the half derivative.

An explanation for the bad half derivative prediction is related to the non-uniqueness of our definition. All our neural network requires is that the operation carried out by the two halves of the network produce the full derivative. Thus any permutation of data points in the middle layer of the network that has a period of 2 will be a possible solution to the middle layer. In other words, while time is imposed on the input and output layers by the training data, there is nothing preventing the network from rearranging time in the middle layer as long as the second network returns time to the right order for the output. As will be shown in the next section, if we regularize the middle layer by penalizing large differences between adjacent middle node values, we obtain much better predictions with only a time reversal symmetry (reflection) left.

[1]All our source files are available at the github repository: `https://github.com/chentan/fractional`.

IV. RESULTS WITH REGULARIZED MIDDLE LAYER

In order to eliminate the the possible period 2 permutations of the nodes, we add to the mean squared error loss function a penalty term that minimizes the square of the difference in values between adjacent nodes in the middle layer. Specifically, the error function is now

$$\begin{aligned} error = \sum_{n=0}^{200} (prediction(n) - exact(n))^2 \\ + \sum_{n=0}^{199} (middle(n) - middle(n+1))^2 \end{aligned} \quad (4)$$

where $middle(n)$ is the value in the nth node in the middle layer (the layer attempting to compute the half derivative). This second term seeks to minimize the difference between adjacent middle node values which will smooth the result. Note that this does *not* require any knowledge of the half derivative and does not use any such knowledge in the training.

With the modified error function given by Equation 4, the network predicts either the half derivative, or a simple reflected symmetry of it. Typical validation result are illustrated in Figures 5 and 6. In both figures, the blue curve is the original function, the purple curve is the exact first derivative. The green curve is the half derivative computed using Equation 2 and the orange curve represents the values of the nodes along the hidden layer of the network. As is clear, now the hidden layer produces a very good estimate of the actual half derivative.

The plots presented are validation results, meaning that the inputs and outputs were not part of the training data set.

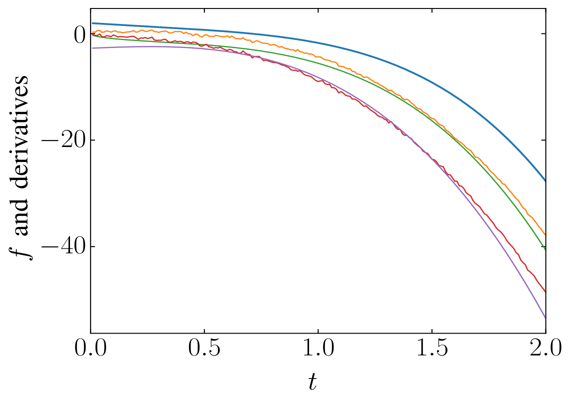

Fig. 6. Typical training result after 2,000 epochs from error function imposing smoothing on middle layer.

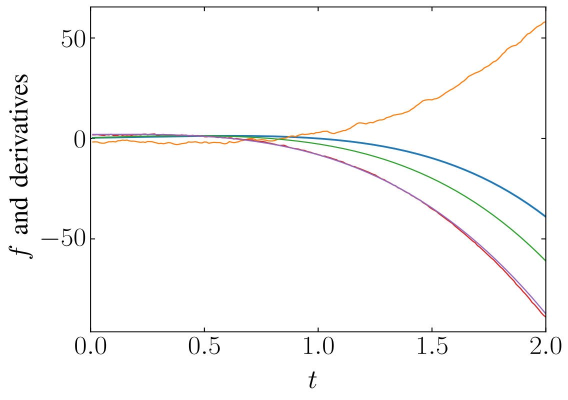

Fig. 7. After 2,000 epochs, convergence of middle layer to -1 times the half derivative.

Also, we emphasize that the training data only included the function and first derivatives (so for training data, only what would correspond to the blue and purple curves), and not the exact half derivative, the green curve.

Regularizing the middle layer by penalizing large value changes between adjacent middle nodes eliminated many of the permutation-type symmetries in the network; however, one still remained, which is multiplication of the half derivative by -1. If half of the network computes -1 times the half derivative, then the whole network will compute the full derivative. So the network will often converge to -1 times the half derivative, as is illustrated in Figure 7. In that figure the predicted half derivative, the orange curve, is the negative of the computed half derivative, which is the green curve.

Because we can compute the exact half derivative, we can compare convergence of the predicted derivative to the full derivative (what the model was trained on) as well as how the middle layer converges to the half derivative. Figure 8 illustrates how the accuracy of the half derivative converges (left figure) and how the input/output training data converges, meaning the full derivative versus the predicted full derivative (right figure).

In the figure, the y-axis is the magnitude of the loss function discussed above. The x-axis are the number of

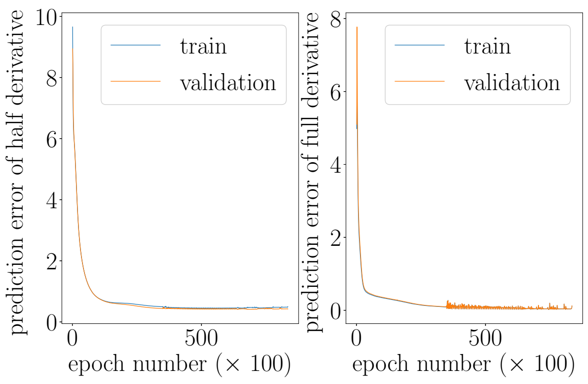

Fig. 8. Convergence of the half derivative (left) and full derivative (right). The blue curves are for the training data, and the orange curves are validation, i.e., untrained, pairs.

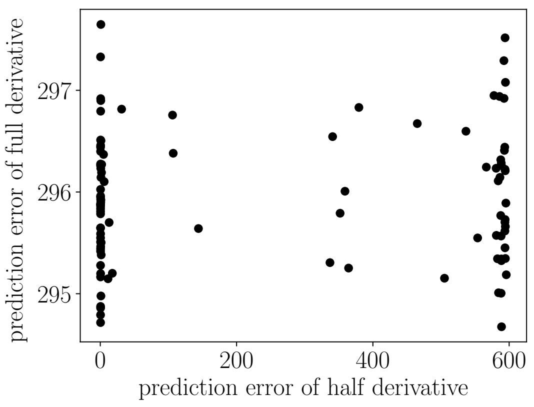

Fig. 9. The x-axis is prediction error of half derivative after full training (5,000 epochs) and the y-axis is the full derivative prediction error at initialization of the training. If the inverse solutions are considered "good" predictions, then the overall number of times a prediction is good is between 80-90%.

training epochs. Note that one epoch means an update of the model with the use of the entire training data set.

Approximately 85-90% of training produces acceptable results, so there is still a significant number of times where the predicted half derivative is not very good. This is not too surprising in some sense because we are obtaining answers that are not directly trained. Figure 9 illustrates the final error for the half derivative prediction versus initial error computations for the full derivative, which can be obtained during training. There are 100 data points, and the x-axis is the final error for the half derivative, and the y-axis is the full derivative prediction error at the initialization of the training.

For the half derivative, about 90% of the points are either with 0 error or a very large error. The vertical set of large error data points corresponds to the inverse solution, so are considered good predictions. The data points in the middle are bad predictions because they seem to bear no resemblance to the half derivative. There are between 10 and 15 points in the middle zone, indicating an overall prediction effectiveness of approximately 85-90%. A typical

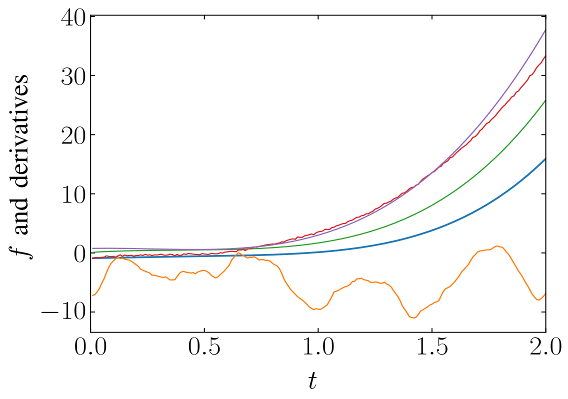

Fig. 10. Example of bad prediction. The prediction is still rather smooth and looks like a function, but has almost no relationship with the half derivative.

"bad" prediction is illustrated in Figure 10.

For training expense, it would typically take approximately 100 minutes to reach 20,000 epochs on a MacBook Pro (2017), 2.3GHz Dual-Core Intel Core i5, memory: 8GB 2133 MHz LPDDR3.

V. CONCLUSIONS AND FUTURE WORK

This paper presented a novel use of a neural network to train a network on easily computed data (first order derivatives) to obtain a much more difficult to compute result (the half derivative). The approach was to force a neural network to have two identical halves. If the full network is trained on the first derivative, then the information in the layer between the identical halves should be the half derivative.

The results are about 80-90% accuracy in producing good half derivative predictions if we include the negative of the half derivative as a good prediction. We emphasize that no half derivative information is used in training.

This paper focuses only on a rather limited set of polynomial inputs, which limits its generalizability. Future work will focus on two areas. The first is training on a more feature-rich set of input functions such as polynomials with many zeros in the domain. This should ensure enough richness in the training data to be able to allow the network to predict half derivatives for a much broader set of functions. The other focus point is on identifying if there is some commonality among the bad predictions to possibly identify if there is a feature of those functions that led to the bad prediction.

ACKNOWLEDGMENT

The authors acknowledge the significant help they received in discussions with Professors Mihir Sen (Aerospace & Mechanical Engineering) and Meng Jiang (Computer Science & Engineering), both from Notre Dame. The original idea for a series of two identical networks as a means to compute a half derivative came up in discussions with Prof Sen, and helpful ideas to extract useful information from between the two networks came from discussions with Prof Jiang.

REFERENCES

[1] Dingyu Xue, Chunna Zhao, and YangQuan Chen, "Fractional order pid control of a dc-motor with elastic shaft: a case study," in *2006 American Control Conference*, 2006, pp. 3182–3187.

[2] T. C. Doehring, A. D. Freed, E. O. Carew, and I. Vesely, "Fractional Order Viscoelasticity of the Aortic Valve Cusp: An Alternative to Quasilinear Viscoelasticity," *Journal of Biomechanical Engineering*, vol. 127, no. 4, pp. 700–708, 01 2005. [Online]. Available: https://doi.org/10.1115/1.1933900

[3] B. Goodwine, "Modeling a multi-robot system with fractional-order differential equations," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 1763–1768.

[4] A.-J. Guel-Cortez, C.-F. Méndez-Barrios, E.-j. Kim, and M. Sen, "Fractional-order controllers for irrational systems," *IET Control Theory & Applications*, vol. 15, no. 7, pp. 965–977, 2021. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/cth2.12095

[5] N. Heymans and J.-C. Bauwens, "Fractal rheological models and fractional differential equations for viscoelastic behavior," *Rheologica Acta*, vol. 33, no. 3, pp. 210–219, 1994. [Online]. Available: https://doi.org/10.1007/BF00437306

[6] K. Leyden and B. Goodwine, "Using fractional-order differential equations for health monitoring of a system of cooperating robots," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 366–371.

[7] K. Leyden, M. Sen, and B. Goodwine, "Large and infinite mass–spring–damper networks," *Journal of Dynamic Systems, Measurement, and Control*, vol. 141, no. 6, Feb 2019, 061005. [Online]. Available: https://doi.org/10.1115/1.4042466

[8] J. Mayes, "Reduction and approximation in large and infinite potential-driven flow networks," Ph.D. dissertation, University of Notre Dame, 2012.

[9] K. Oldham and J. Spanier, *The Fractional Calculus Theory and Applications of Differentiation and Integration to Arbitrary Order*. Elsevier Science, 1974.

[10] M. D. Ortigueira, *Fractional Calculus for Scientists and Engineers*, ser. Lecture Notes in Electrical Engineering. Netherlands: Springer Netherlands, 2011, vol. 84.

[11] A. Oustaloup, *La d ́erivation non enti‘ere*. Hermes, 1995.

[12] J. Sabatier, "Beyond the particular case of circuits with geometrically distributed components for approximation of fractional order models: Application to a new class of model for power law type long memory behaviour modelling," *Journal of Advanced Research*, vol. 25, pp. 243–255, 2020, recent Advances in the Fractional-Order Circuits and Systems: Theory, Design and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2090123220300667

[13] H. Sun, Y. Zhang, D. Baleanu, W. Chen, and Y. Chen, "A new collection of real world applications of fractional calculus in science and engineering," *Communications in Nonlinear Science and Numerical Simulation*, vol. 64, pp. 213–231, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1007570418301308

[14] D. Valério and J. S. de Costa, *An Introduction to Fractional Control*. London, United Kingdom: Institution of Engineering and Technology, 2013.