10 January 2025, Preprint v1 · CC-BY 4.0 Qeios PREPRINT

Research Article

A Quasilinear Algorithm for Computing Higher-Order Derivatives of Deep Feed-Forward Neural Networks

Kyle R. Chickering[1]

1. Department of Mathematics, University of California, Davis, United States

The use of neural networks for solving differential equations is practically difficult due to the exponentially increasing runtime of autodifferentiation when computing high-order derivatives. We propose n-TANGENTPROP , the natural extension of the TANGENTPROP formalism[1] to arbitrarily many derivatives. n-TANGENTPROP computes the exact derivative $d^n/dx^n f(x)$ in quasilinear, instead of exponential time, for a densely connected, feed-forward neural network f with a smooth, parameter-free activation function. We validate our algorithm empirically across a range of depths, widths, and number of derivatives. We demonstrate that our method is particularly beneficial in the context of physics-informed neural networks where n-TANGENTPROP allows for significantly faster training times than previous methods and has favorable scaling with respect to both model size and loss-function complexity as measured by the number of required derivatives. The code for this paper can be found at https://github.com/kyrochi/n_tangentprop.

Corresponding author: Kyle R. Chickering, krchickering@ucdavis.edu

I. Introduction

Physics-informed neural networks (PINNs) were introduced as a numerical method to solve forward and inverse problems involving differential equations using neural networks instead of traditional numerical solvers[2]. Their use has recently come under scrutiny for several reasons, including a lack of high-accuracy results, poor run-times compared to standard numerical methods, and complicated training dynamics[3]. Due to methodological issues in the aforementioned studies, including the failure to use the well-established strategies outlined in[4], we believe that the pursuit of PINNs

should not be abandoned and to that end, propose an algorithm which partially addresses the valid concerns over PINN training times.

A primary reason to use PINNs over standard numerical methods is because by approximating the solution to an ODE or PDE by a neural network we obtain a C^∞ approximation to the model solution which can be evaluated at arbitrary points in the domain, as well as allowing for the study of high-order derivatives (see Figures 7 and 10). This strength is also a weakness, since during training we must repeatedly take derivatives of the neural network with respect to the inputs. This is done using autodifferentiation[5][2], which suffers from unfavorable scaling when taking multiple derivatives. In particular, taking n derivatives of a neural network f with M parameters gives the exponential runtime $\mathcal{O}(M^n)$. For training PINNs we often need to take two or more derivatives, and in many practical applications this exponential runtime already becomes prohibitive. Furthermore this difficulty cannot simply be overcome by horizontally scaling compute, since repeated applications of autodifferentiation cannot be parallelized on a GPU due to the recursive nature of computing high-order derivatives.

In this paper we introduce n-TANGENTPROP , which addresses these issues by computing $\frac{d^n}{dx^n} f(x)$ in quasilinear $\mathcal{O}(e^{\sqrt{n}} M)$ time instead of the exponential time $\mathcal{O}(\frac{e^{\sqrt{n}}}{n} M^n)$. n-TANGENTPROP is an exact method, and thus there is no accuracy degradation when using this method. n-TANGENTPROP is the natural extension of the TANGENTPROP formalism[1] to n derivatives. TANGENTPROP was introduced in the context of MNIST digit classification as a way to enforce a smoothness condition on the first derivative of a neural network based classifier. The motivation for n-TANGENTPROP starts from the observation that for PINN training we only need higher-order derivatives with respect to the network inputs, not with respect to the network weights. In practice the dimensionality of the input data d is much smaller than the number of parameters M, and therefore it is unnecessary to compute a fully filled out computational graph for all higher order derivatives. Instead of building the full computational graph, we directly differentiate the network during the forward pass and can thus compute all n derivatives with respect to the network inputs during a single forward pass.

Our contributions are three-fold

1. We derive the n-TANGENTPROP formalism and give an algorithm for it's implementation.
2. We show that for simple network architectures consisting of stacked linear, densely connected layers with the $\tanh$ activation function, our method empirically follows the theoretical scaling

laws for a variety of widths, depths, and batch sizes.

3. We show that in the context of PINN training our method can significantly reduce training time and memory requirements when compared to the standard PINN implementation.

II. PINN Training

PINNs are neural networks trained to approximate the solution to a given ODE or PDE[2]. Because neural networks with smooth activation functions are C^∞ function approximators, and C^∞ functions are dense in most function space which are used in practice (like the L^2 based Sobolev family of function spaces), we can train a neural network using gradient descent to approximate the solution to a given differential equation. It is this simple observation that led to the introduction of PINNs in the paper[2]. We give a brief overview of the methodology behind PINNs, but refer the reader to the recent surveys[6][7][8] and the references therein, as the literature abounds with introductory material on PINNs.

Let $u_\theta(\mathbf{x})$ be a feed-forward, densely connected, neural network with parameters θ. Our goal is to optimize these parameters in such a way that u_θ is an approximate C^∞ solution to the differential equation $F(\partial^\alpha u; \mathbf{x}) = 0$ for some multi-index α[9]. We train the neural network on the discrete domain $\Omega = \{\mathbf{x}_1, \cdots \mathbf{x}_N\}$ using the loss target

$$L(u_\theta) = \frac{1}{N} \sum_{k=1}^{N} |F(\partial^\alpha u_\theta; \mathbf{x}_k)|^2 + \mathrm{BC}, \tag{1}$$

which is the mean-squared error (MSE) of the differential equation residual with BC being appropriately enforced boundary conditions. Since u_θ is $|\alpha|$-times continuously differentiable we can use autodifferentiation to exactly compute the derivatives appearing above, and thus by the approximation theorem[10][11], if the solution to $F(\partial^\alpha u; \mathbf{x}) = 0$ lies in a space in which C^∞ functions are dense, we can theoretically train a neural network to approximate the true solution of the PDE to arbitrary precision.

In practice this is not so easy: PINN training is complicated by needing to enforce boundary conditions, which introduces problems inherent to multi-target machine learning[12][13]. There is also the problem of effectively choosing collocation points from the domain, as well as the problem of choosing a good network architecture[4][14]. PINN training appears naturally unstable, likely due to a poorly conditioned Hessian, and thus training these networks can be difficult[15][16][17]. This

instability has further been related to the conditioning of a specific differential operator related to the underlying differential operator[18]. PINNs also appear to struggle fitting high-frequency components of the target solution[19] as a consequence of spectral bias (f-principle)[20][21].

Additionally, convergence under the loss function (1) is often slow. In practice it is usually better to use "Sobolev training"[22][23][24] which replaces (1) with the Sobolev-norm[9] loss function

$$L^{(m)}_{\text{Sobolev}}(u_\theta) = \frac{1}{N}\sum_{k=1}^{N}\sum_{j=0}^{m} Q_j \left|\nabla^j_{\mathbf{x}} F(\partial^\alpha u_\theta; \mathbf{x}_k)\right|^2 + \text{BC}, \tag{2}$$

where Q_j are relative weights which add additional hyperparameters to the training[23]. While this loss function generally improves accuracy, it also requires computing m extra derivatives of the neural network u_θ. Due to the nature of autodifferentiation, this trade-off quickly becomes costly and in practice we can often only train with $m = 1$ or $m = 2$, despite the fact that higher m usually results in better solution accuracy. n-TANGENTPROP makes this trade-off much cheaper and we hope that future authors are able to train with $m = 4$ or higher while retaining reasonable training times.

The appearance of high-order derivatives is also not uncommon in PINN applications. Wang et al. [25] show that to satisfactorily compute successive high-order unstable shock profiles for the Burgers, De Gregorio, and Boussinesq equations, one must take high-order derivatives. For example, to compute the m-th smooth, self-similar shock profile for Burgers equation one must take $2m + 3$ derivatives. The authors compute the first and second profiles which already requires taking five derivatives and is already slow. Using n-TANGENTPROP we are able to compute the third and fourth profiles in this paper in a reasonable amount of time (See section IV-C1 below).

To our knowledge there has been no prior work which directly addresses the exponential runtime of autodifferentiation in the literature. Instead, works aimed at making training more computationally efficient rely on augmenting the PINN training with some sort of numerical differentiation[26][27], pre-training or transfer learning methods[28][29], or efficient sampling of the collocation points[30]. There are too many articles in this direction to compile an exhaustive list, and we instead refer again to the many surveys and studies which abound[6][7][8][4]. We stress that n-TANGENTPROP is an exact method and all of the aforementioned studies would be accelerated by adopting our proposed methodology.

doi.org/10.32388/92OY7E

III. n-TANGENTPROP

A. Autodifferentiation

Autodifferentiation is a method for computing the derivatives of a neural-network using the network's computational graph[5][31]. Autodifferentiation is usually applied in the context of gradient descent for optimizing neural networks where it is used to efficiently and exactly compute the first partial derivatives of the loss with respect to each of the network weights[31]. It is usually applied once per training iteration, and outputs a vector of first-order partial derivatives with respect to all network inputs (including the weights). It is able to do this computation in $\mathcal{O}(M+d)=\mathcal{O}(M)$ time, where M is the number of model parameters and d is the dimensionality of the input.

It is less common for autodifferentiation to be applied repeatedly. However this repeated application is the key to the effectiveness of PINNs[2]. Because there are $\mathcal{O}(M^2)$ second-order partial derivatives, and $\mathcal{O}(M^n)$ n-th order partial derivatives, the repeated application of auto-differentiation takes $\mathcal{O}(M^n)$ to compute all n-derivatives. This bound does not fully account for the amount of time required to take the n-th derivative of the activation function, see below. The full runtime is $\mathcal{O}\left(\frac{e^{\sqrt{n}}}{n}M^n\right)$. For small networks and a low number of derivatives this runtime is acceptable, but the exponential growth makes training large PINN models or PINN models for equations involving high-order derivatives prohibitively difficult. Furthermore, as discussed above, it appears beneficial to use the Sobolev loss (2) which requires taking even more derivatives. In practice PINN training becomes prohibitively slow when computing more than three or four derivatives of the network.

It is easy to see why autodifferentiation is not the right tool for computing derivatives in the PINN loss function: we do not need every partial derivative computed by autodifferentiation. In fact, we only need the n-th partial derivatives corresponding to the network inputs, i.e.

$$\nabla_{\mathbf{x}}^n f(\mathbf{x}) := \frac{\partial^n}{\partial \mathbf{x}^n} f(\mathbf{x}),$$

which is often a sparse subset of the total derivatives computed by auto-differentiation. In what follows we propose n-TANGENTPROP , which is an efficient quasilinear algorithm to compute only these partial derivatives that are needed for the PINN loss function.

B. Tangent Prop and n-TANGENTPROP

The TANGENTPROP formalism introduced in[1] derives the exact forward and backward propagation formulas for the derivative of a deep feed-forward network. This was done in the context of MNIST digit classifictation based on the observation that the classifier (the neural network) should be invariant to rotations of the input digits, and thus the derivative of the classifier with respect to the inputs (the tangent vector) should be zero. This effectively enforces a first-order (derivative) constraint on the classifier.

Let σ be an activation function, $\mathbf{a}^\ell$ be the activations at the ℓ-th layer, $\mathbf{w}$ be the weight matrix, and $\mathbf{x}^0$ be the network inputs, the authors in[1] derive the formula for the first derivative γ of the network using a single forward pass

$$a_i^\ell = \sum_j w_{ij}^\ell x_j^{\ell-1}, \quad x_j^{\ell-1} = \sigma(a_j^{\ell-1}), \tag{3a}$$

$$\gamma_i^\ell = \sum_j w_{ij}^\ell \xi_j^{\ell-1}, \quad \xi_j^{\ell-1} = \sigma'(a_j^\ell)\gamma_j^\ell. \tag{3b}$$

The formula (3b) is derived by applying the chain rule to the per-layer activation formula (3a).

We can naturally extend TANGENTPROP to compute n derivatives in a single forward pass by applying Faà di Bruno’s formula[32], which generalizes the chain rule to multiple derivatives. Faà di Bruno’s formula states that for the composition of C^n continuous functions f and g we have

$$f(g(x))^{(n)} = \sum_{\mathbf{p}\in\mathcal{P}(n)} C_{\mathbf{p}}(f^{|\mathbf{p}|}(g(x)) \prod_{j=1}^{n} (g^{(j)}(x))^{p_j}, \tag{4}$$

where the sum is taken over the set $\mathcal{P}(n)$ of partition numbers of order n, which consists of all tuples $\mathbf{p}$ of length n and satisfying $\sum_{j=1}^n jp_j = n,\ 0 \le p_j \le n,$ and $|\mathbf{p}| = \sum_j p_j$. The constants $C_{\mathbf{p}}$ are explicitly computable and the size of the set $\mathcal{P}$ is found using the partition function $p(n) = |\mathcal{P}(n)|$ whose combinatorial properties are well-studied[33].

Since neural networks with smooth activation functions are C^∞, we can apply our new formalism, which we call n-TANGENTPROP , to take arbitrarily many derivatives of deep feed-forward neural-networks. Using Faà di Bruno’s formula (4) in place of the chain rule in (3a) allows us to compute an arbitrary derivative in the same forward pass that we compute the activations. The formula for the n-th derivative $\gamma^{(n)}$ is given by

doi.org/10.32388/92OY7E

$$\gamma_i^\ell = \sum_j w_{ij}^\ell \xi_j^{\ell-1}, \qquad (5a)$$

$$\xi_j^{\ell-1} = \sum_{\mathbf{p}} C_{\mathbf{p}} \sigma^{(|\mathbf{p}|)}(a_j^\ell) \prod_{m=1}^{n} (\xi_j^{(m)})^{p_m}. \qquad (5b)$$

where a_j^ℓ are the forward activations computed in (3a). Thus, in a single forward pass through the model we can compute all of the required derivatives at once with runtime of $\mathcal{O}(np(n)M)$, where $p(n)$ is the partition function. Thus, we have reduced the exponential runtime from auto-differentiation to a quasilinear runtime (see Algorithm III-B and the tighter bound derived below). Note that the derivatives must be computed in order, since $\xi^{(m)}$ depends on $\xi^{(k)}$ for all $k < m$.

The coefficients $C_{\mathbf{p}}$ appearing in (5b) are the coefficients of the Bell polynomials of the second kind (See[32] and references therein). These are well-studied and explicitly computable, and for efficiently implementing Algorithm III-B we recommend pre-computing and caching the required coefficients (see our implementation code for more details).

doi.org/10.32388/92OY7E

Algorithm 1 Forward Pass with Higher-Order Derivatives

1: **procedure** FORWARD(x, n) ▷ x: input, n: derivative order
2: **if** $n = 0$ **then**
3: **for** $layer$ in NETWORK **do**
4: $x \leftarrow layer(\sigma(x))$ ▷ σ is activation function
5: **end for**
6: **return** x
7: **end if**
8: $y \leftarrow$ array of length $n + 1$
9: $y_0 \leftarrow L_1(x)$ ▷ L_1 is first layer
10: $y_1 \leftarrow L_1(\mathbf{1}) - b_1$
11: **for** $i \leftarrow 2$ to n **do**
12: $y_i \leftarrow L_1(\mathbf{0}) - b_1$
13: **end for**
14: **for** L in NETWORK[2 :] **do**
15: $a \leftarrow \sigma(y_0, n)$
16: **for** $i \leftarrow n$ down to 1 **do**
17: $z \leftarrow \mathbf{0}$
18: **for** (c, e) in BELL[i] **do**
19: $s \leftarrow \sum e$
20: $t \leftarrow \prod_{j:e_j \neq 0} y_j^{e_j}$
21: $z \leftarrow z + c \cdot t \cdot a_s$
22: **end for**
23: $y_i \leftarrow z$
24: **end for**
25: $y_0 \leftarrow L(a_0)$
26: **for** $i \leftarrow 1$ to n **do**
27: $y_i \leftarrow L(y_i) - b$ ▷ Subtract bias
28: **end for**
29: **end for**
30: **return** y
31: **end procedure**

For the sake of completeness we give a tighter bound on the runtime which takes into account the dependence on n of the summation appearing in (4). The combinatorial properties of the summation over the integer partitions $\mathcal{P}(n)$ are well studied. In particular, the partition function $p(n)$ counts the number of integer partitions and thus $p(n) = |\mathcal{P}(n)|$. A well-known and classical result of Hardy and Ramanujan[34] provides an upper and lower bound on the partition function $p(n)$ which then yields the asymptotic behavior

$$p(n) = \mathcal{O}\left(\frac{e^{\sqrt{n}}}{n}\right).$$

This implies the more refined runtime depending on n and M of $\mathcal{O}(np(n)M) \sim \mathcal{O}(e^{\sqrt{n}}M)$ which is our claimed quasilinear bound. Note that during autodifferentiation the Faà di Bruno formula must implicitly be applied to the activation function σ and therefore autodifferentiation has an actual asymptotic runtime of $\mathcal{O}\left(\frac{e^{\sqrt{n}}}{n}M^n\right)$. Finally we remark that the memory complexity of n-TANGENTPROP is linear at $\mathcal{O}(nM)$ (without a modification from $p(n)$) while the memory complexity of autodifferentiation is exponential at $\mathcal{O}(M^n)$. Thus, not only does our algorithm compute derivatives faster than autodifferentiation, but we can compute more derivatives on the same hardware than is even possible using autodifferentiation.

IV. Experiments

We begin by demonstrating that for a wide range of feed-forward neural network architectures that our proposed method indeed follows the theoretical asymptotic scaling laws. In particular we consider a standard feed-forward network with uniform width across the layers and the $\tanh$ activation function. Then we use our method to train a PINN model to compute the smooth stable and unstable profiles for the self-similar Burgers profile using the methodology proposed in[25]. This problem requires taking a large number of derivatives for the training to converge, and we demonstrate that our method is able to break through the computational bottlenecks imposed by autodifferentation and we are able to compute higher-order profiles which were previously either impractical or impossible to compute using autodifferentiation on a single GPU.

A. Implementation Details and Methodology

We run our experiments using Python and PyTorch[31]. In particular we implement n-TANGENTPROP as a custom forward method for a PyTorch torch.nn.Module implementation of a deep feed-forward network. For the PINN experiments we then build a custom PINN training framework to handle the PINN training loop, and we use an open-source L-BFGS implementation[35] instead of the PyTorch L-BFGS due to the latter not supporting line-search[1]. All experiments were run locally on a single NVIDIA A6000 GPU.

doi.org/10.32388/92OY7E

B. Forward-Backward Pass Times

PyTorch implements several asynchronous optimizations that make benchmarking difficult. To mitigate the effects of built-in optimizations on our benchmarking we implement the following steps

1. Randomly shuffle the experiments over all parameters to ensure that execution order is not a factor in the results.
2. Synchronize CUDA between runs.
3. Enable cudnn.benchmark.
4. Run the Python garbage collector between runs.
5. Use the Python performance counter instead of the timing module.

These mitigation strategies allow for a fairer comparison between autodifferentiation and n-TANGENTPROP .

We first explore the effect of n-TANGENTPROP on the computation of a single forward and backward pass to verify that the empirical performance aligns with the predicted theoretical performance suggested by our derivation. In particular we would expect to see exponential run-times for autodifferentiation and quasilinear run-times for n-TANGENTPROP .

For a given network we compute and time the forward pass through the network, compute the loss outside of a timing module, then compute and time a backwards pass through the network. The total time includes the time it takes to compute the loss function, while the forward and backwards pass times only include the time it takes to compute the given pass.

For a fixed network size, we find that the end-to-end times for a combined forward and backward pass for autodifferentiation scales exponentially and that n-TANGENTPROP scales roughly quasilinearly (Figure 1), keeping with the theoretical predictions made above by our formalism. We can further decompose this total execution time into its forward and backward times (Figure 2 and Figure 3 respectively). We see that the n-TANGENTPROP formalism gives more significant performance gains during the forward pass when compared to the backwards pass. We hypothesize that this is due to PyTorch graph optimizations that are applied automatically to the autodifferentiation implementation and are not included in our n-TANGENTPROP implementation. This difference is seen most plainly in Figure 3, where autodifferentiation outperforms n-TANGENTPROP in backwards pass times for small numbers of derivatives.

doi.org/10.32388/92OY7E

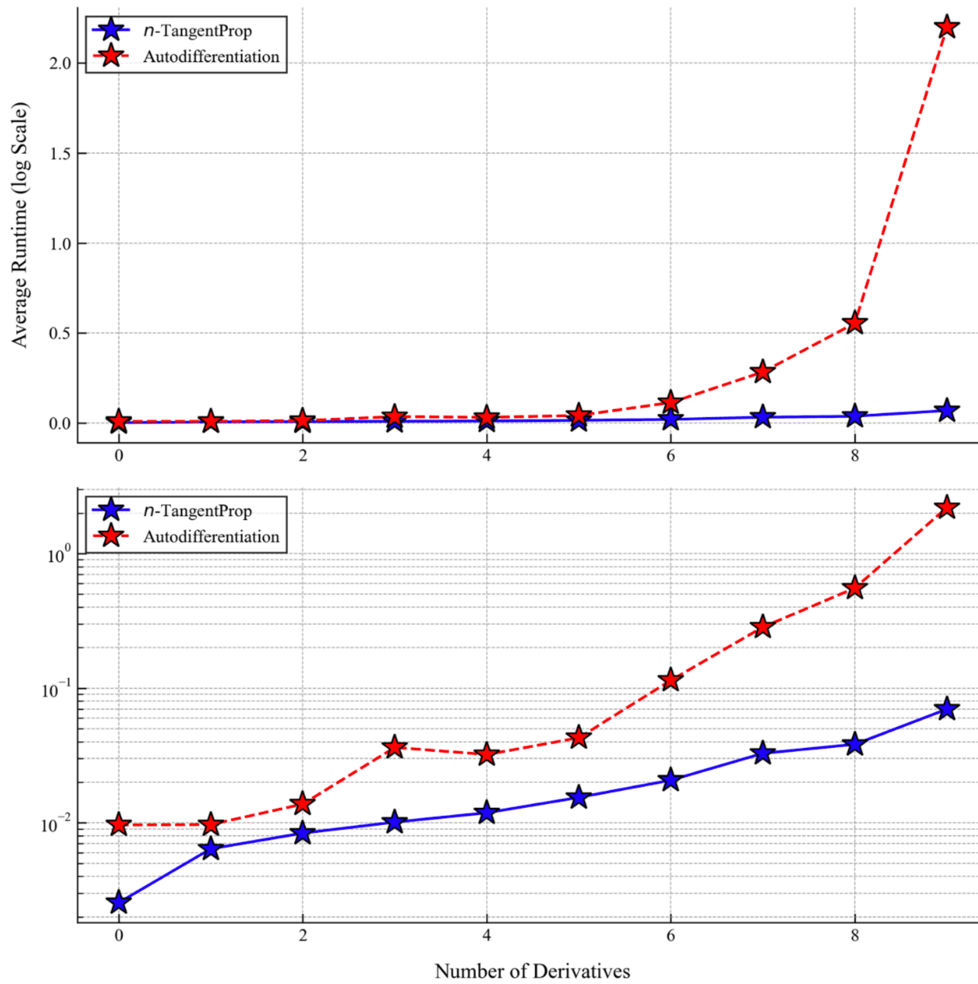


Figure 1. Average runtime for a combined forward and backwards pass using autodifferentiation (red) and n-TANGENTPROP (blue). The top and bottom frames show the same data, however the bottom frame is plotted with a logarithmic y-axis. Each model is run 100 times and the average for each trial is plotted. The network has 3 hidden layers of 24 neurons each, a common PINN architecture. The batch size is $2^8 = 256$ samples. The forward and backwards pass times are shown separately in Figures 2 and 3 respectively.

doi.org/10.32388/92OY7E

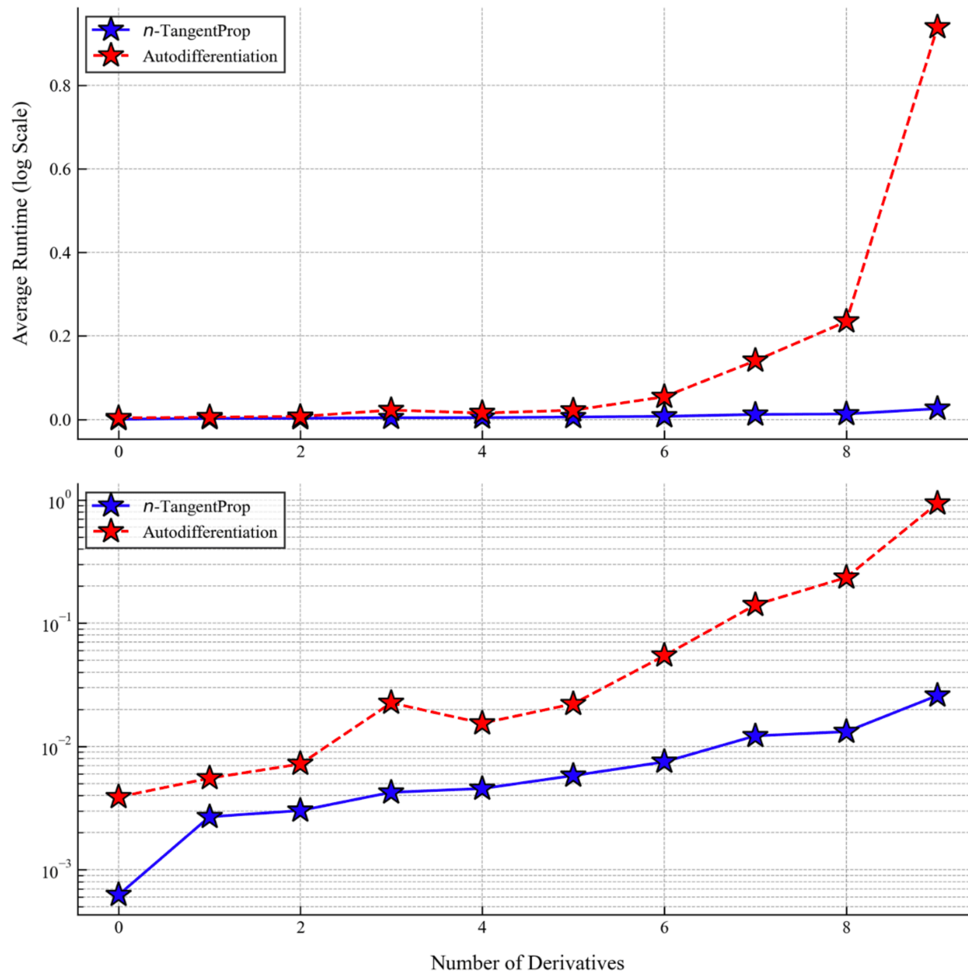


Figure 2. Forward pass times for the model shown in Figure 1. The top and bottom frames show the same data, however the bottom frame is plotted with a logarithmic y-axis. Each model is run 100 times and the average for each trial is plotted. The network has 3 hidden layers of 24 neurons each, a common PINN architecture. The batch size is $2^8 = 256$ samples.

doi.org/10.32388/92OY7E

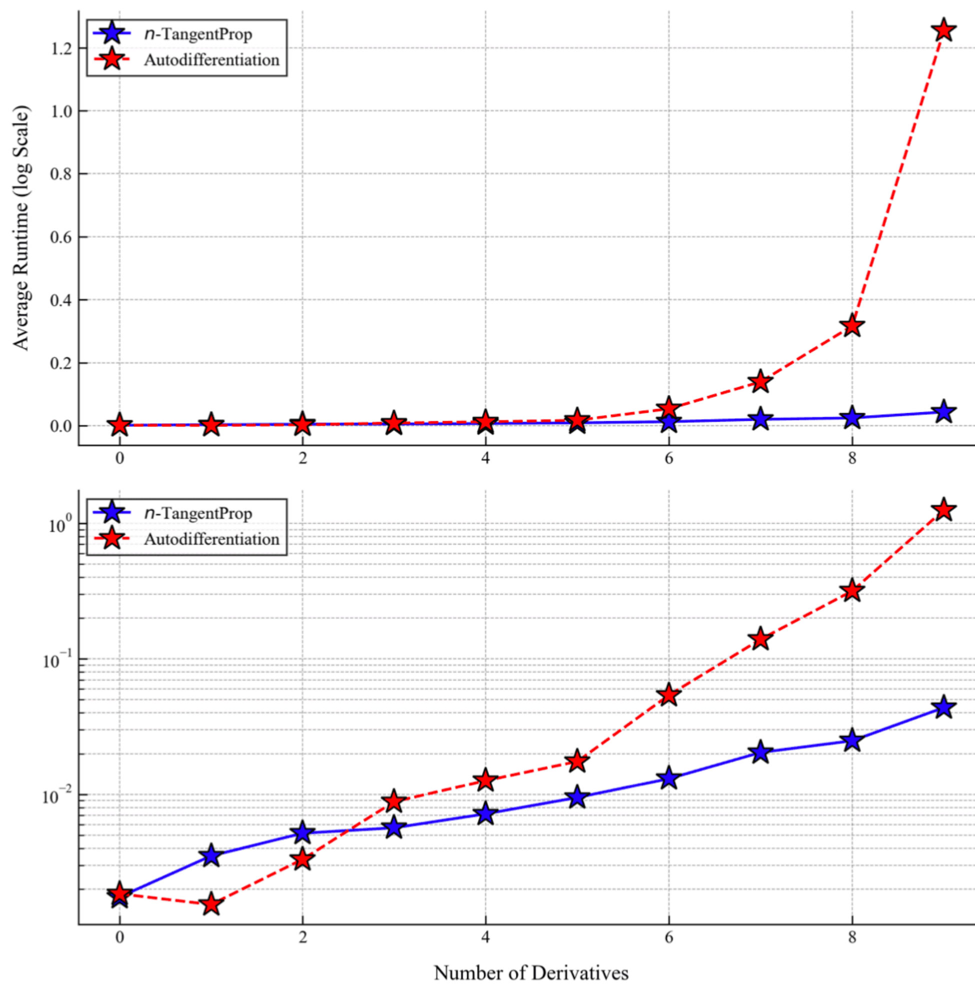


Figure 3. Backwards pass times for the model shown in Figure 1. The top and bottom frames show the same data, however the bottom frame is plotted with a logarithmic y-axis. Each model is run 100 times and the average for each trial is plotted. The network has 3 hidden layers of 24 neurons each, a common PINN architecture. The batch size is $2^8 = 256$ samples.

We run extensive experiments to analyze the effect of varying batch size, network width, network depth, and number of derivatives. The results of the forward passes are summarized in Figure 4, and the results of the combined forward and backward pass are summarized in Figure 5.

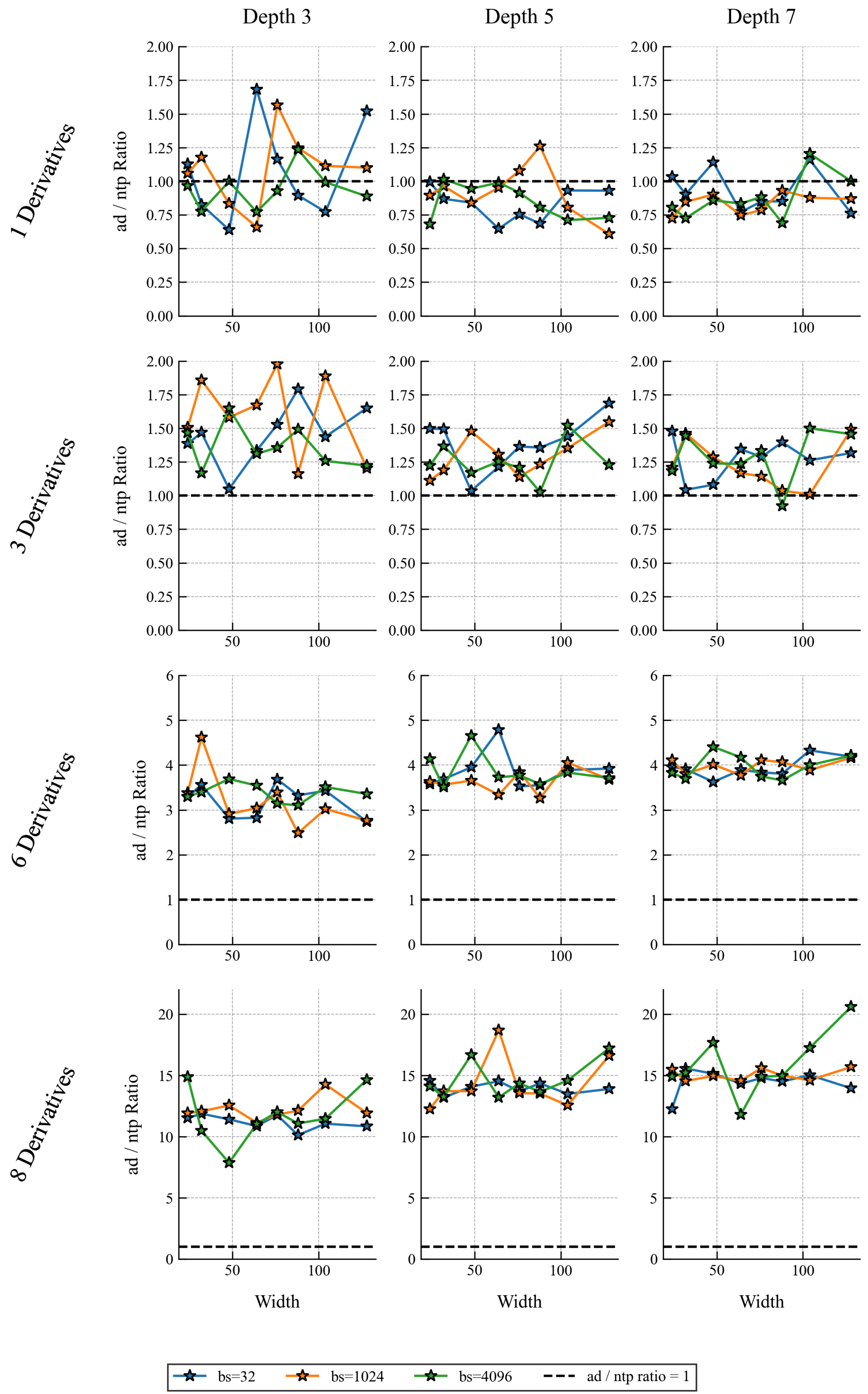


Figure 4. The ratio of forward pass run times between autodifferentiation and n-TANGENTPROP for a variety of network architectures, input batch sizes, and number of derivatives. A ratio greater than 1 indicates that n-TANGENTPROP was faster than

autodifferentiation. The baseline ratio of 1 is plotted as a horizontal dashed line. All plotted data points represent the average of 100 trials.

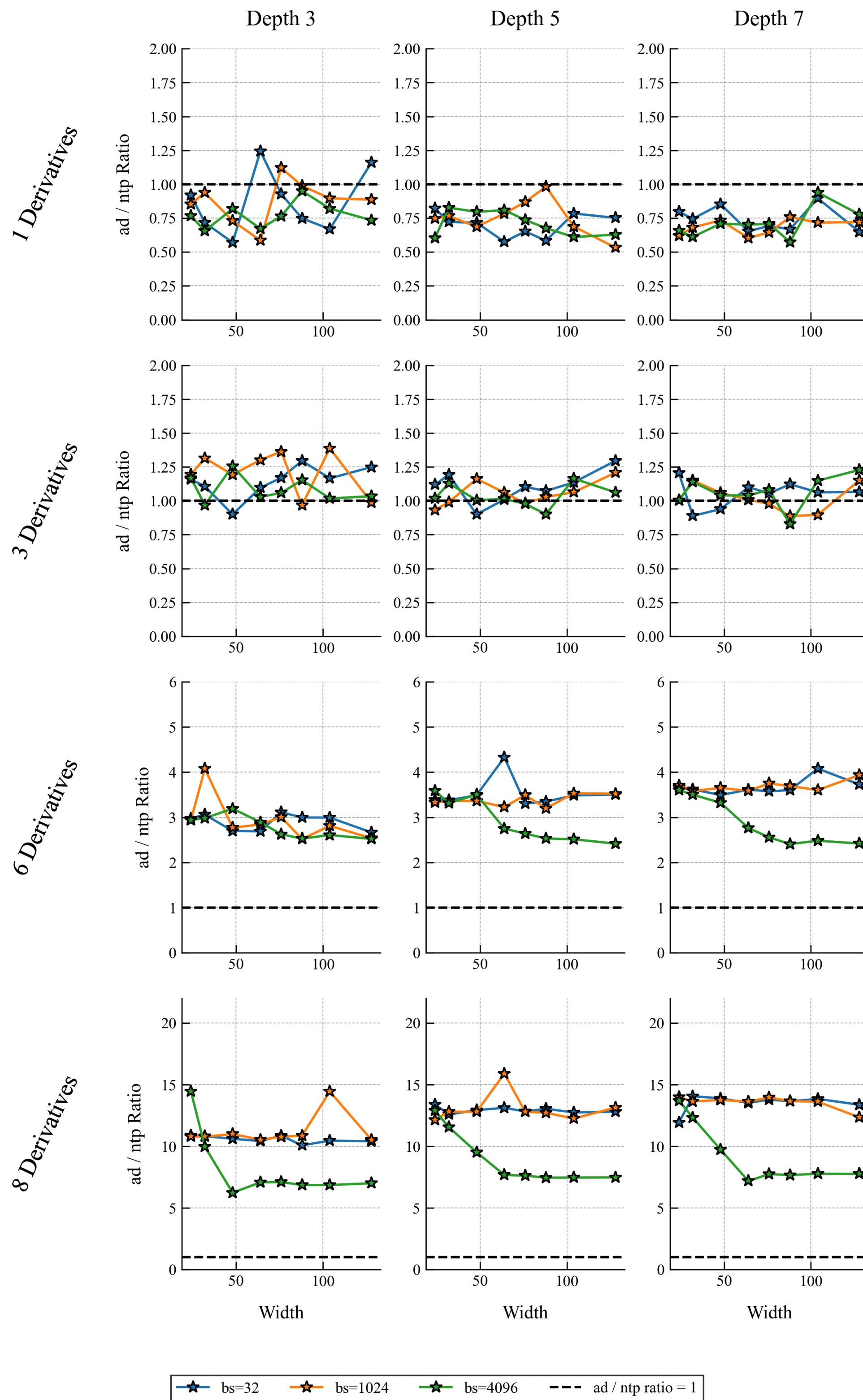


Figure 5. The ratio of combined forward-backward pass run times between autodifferentiation and n-TANGENTPROP for a variety of network architectures, input batch sizes, and number of derivatives. A ratio greater than 1 indicates that n-

TANGENTPROP was faster than autodifferentiation. The baseline ratio of 1 is plotted as a horizontal dashed line. All plotted data points represent the average of 100 trials. The forward pass time ratio alone is plotted in Figure 4.

We point out several salient features in these results. First, we observe a performance gap between n-TANGENTPROP and autodifferentiation for low derivatives. This is likely a consequence of implementation details, rather than a deficiency with the proposed methodology. The PyTorch implementation of autodifferentiation is heavily optimized for execution on a GPU, and while our implementation makes attempts at closing this performance gap, it is written in Python, rather than a lower level language such as C++, and lacks sophisticated optimization strategies. We suspect that a more refined implementation would close the gap seen for low derivatives.

Second, we observe that the performance gains afforded by n-TANGENTPROP decrease as we increase the batch size. We hypothesize that this effect is also due to a lack of optimization to take full advantage of the parallelized computational ability of the GPU in our implementation. For example, our implementation does not fully vectorize the computation of Equation 5b and thus does not take full advantage of the hardware scaling afforded by our GPU. Similarly, we observe that the performance gains from n-TANGENTPROP decrease as we increase the network width. We suspect that this is also a consequence of the lack of vectorization. Increasing either width or batch size scales the compute horizontally, and we have not fully optimized our implementation to account for this horizontally scaled compute.

Third, we observe that for all of the tested derivatives and batch-sizes, the standard PINN architecture of three hidden layers and 24 neuron widths[2] performs better with n-TANGENTPROP than autodifferentiation, at least for derivatives of order three or higher. This suggests that for PINN problems involving higher-order derivatives, n-TANGENTPROP can be used as a drop in replacement without any further implementation tuning (See Section IV-C below).

Finally, we observe an apparent asymptote for the combined forward-backward pass times as the number of parameters and batch size increase (see the bottom rows of Figure 5). We suspect that this is because as we increase the relative number of FLOPs the theoretical gains from n-TANGENTPROP begin to dominate the superior optimization in PyTorch's autodifferentiation implementation. We hypothesize that with a stronger n-TANGENTPROP implementation we would begin to see similar asymptotic behaviors even when taking fewer derivatives.

doi.org/10.32388/92OY7E

We add that we could not compute more than nine derivatives using autodifferentiation because the required memory exceeded the 49 GB of memory available on our GPU.

C. End-to-End PINN Training

Forward-Backward pass times should correlate to end-to-end model training but it is still important to measure the effect of the proposed modifications over the long time-horizons and multiple optimizers present in end-to-end PINN training. For example, our proposed n-TANGENTPROP method uses a different memory footprint for forward pass than autodifferentiation does. It can be difficult to reason theoretically about the effect that such changes will have to the end-to-end performance of machine learning models, and as such, empirical analysis is imperative to rule out performance degradation which arises as a consequence of the complicated end-to-end training testbed.

We find that the widespread use of the L-BFGS optimizer adds to the improvements afforded by n-TANGENTPROP . L-BFGS performs quasi-second order optimization using a line-search[36] which requires performing multiple forward passes through the network but only a single backwards pass. Thus, the forward pass performance seen in Figure 4 is expected to dominate and we expect that our unoptimized n-TANGENTPROP algorithm will outperform standard PINN implementations using autodifferentiation.

1. Unstable Self-Similar Burgers Profiles

Burgers equation is the canonical model for 1D shock formation phenomenon[37] and is given by the PDE

$$\partial_t u + u\partial_x u = 0. \tag{6}$$

Due to the nonlinear steepening of the wave profiles leading ultimately to a gradient blowup, studying the behavior of shock solutions near the singularity is difficult. Work in the 20th century explored the use of self-similarity to study the breakdown of smooth solutions near a shock[38]. Under the self-similar coordinate transformation

$$u(x,t) = (1-t)^{\lambda} U(X), \quad X = x(1-t)^{-1-\lambda},$$

the PDE (6) becomes the ODE[39] for U in X

$$-\lambda U + (1+\lambda)X + U)U' = 0, \tag{7}$$

for a scalar valued parameter $\lambda \in \mathbf{R}^{>0}$. While the solution of this particular problem is elementary and given implicitly by

$$X = -U - CU^{1+\frac{1}{\lambda}}, \tag{8}$$

the techniques used for the numerical analysis and solution of this problem can be applied to more challenging problems to yield highly non-trivial results about shock formation in complicated nonlinear equations[25].

From (8) we observe that the solution U will be smooth whenever $1 + \frac{1}{\lambda}$ is an integer, and be physically realizable (odd) whenever $1 + \frac{1}{\lambda} = 2k$ for some positive integer k [39]. Thus, the possible values of λ corresponding to smooth solutions are $\lambda = \frac{1}{2k}$ for $k = 1, 2, \cdots$. For all other values of λ the solution U will suffer a discontinuity at the origin in one of it's higher-order derivatives.

Our goal for this problem is to find these physically realizable solutions, a problem which is complicated by the fact that the profiles corresponding to $k = 2, 3, \cdots$ are physically and numerically unstable[39]. Traditional solvers will not converge to these solutions, and they do not manifest as real-world shocks due to a collapse towards the solution corresponding to $k = 1$. Regardless, these unstable profiles are important to understand mathematically and can give insights into the underlying behavior of certain systems. See for example the papers[40][41], which apply self-similar methodology to perturbations of the self-similar Burgers equation 7.

Wang et al.[25] propose a methodology for solving Equation 7 for an unknown value of λ using PINNs to perform a combined forward-inverse procedure and simultaneously solve for U and λ. Such a methodology demonstrates the advantage that PINNs have in certain numerical settings, since solving this problem using traditional solvers is challenging. We refer the reader to the study by Biasi[42] which addresses a similar problem using traditional numerical methods and highlights the attendant difficulties.

The primary observation in[25] is that a solution to (7) is smooth for all values of λ, except at the origin, where a discontinuity will appear for derivatives $n \geqslant 1 + \frac{1}{\lambda}$. Thus, if we restrict the value of λ to $[1/3, 1]$ and enforce a smoothness condition on the third derivative of our neural network, we will converge to the unique smooth profile (if one exists) in the range $[1/3, 1]$. This is because for any non-smooth profile in this range, the third derivative or lower must be non-smooth at the origin. To find higher-order smooth profiles we can look between $[1/(2k+3), 1/(2k+1)]$ and enforce a smoothness condition on the $2k+3$-th derivative.

Using a PINN, we can enforce differentiability at the origin by taking sufficiently many derivatives there, since the neural network solution is smooth. This forces the solution to be smooth, which in turn gives gradient signal to push λ towards a valid value. We loosely follow the training schedule used in[25] to compute the first, second, third, and fourth profiles in quasilinear, rather than exponential time. The authors in[25] only computed the first and second profiles, so our results represent the first time the third and fourth profiles have been computed using PINNs. We stress that computing these solutions requires taking many derivatives and thus the n-TANGENTPROP formalism is well-suited for this type of problem. Additionally, computing the third or higher profile is extremely computationally intensive.

We note in passing that we were unable to reproduce the accuracy results claimed by the Wang et al. paper[25] and the authors do not provide access to their code. Regardless, our work is orthogonal to theirs and any future attempts at a reproduction of their work will benefit from using n-TANGENTPROP .

We run our self-similar Burgers experiments using 64-bit floating point precision. We give a detailed description of our methodology and results below in Appendix -A, which we hope will contribute to the reproducibility of the[25] results.

doi.org/10.32388/92OY7E

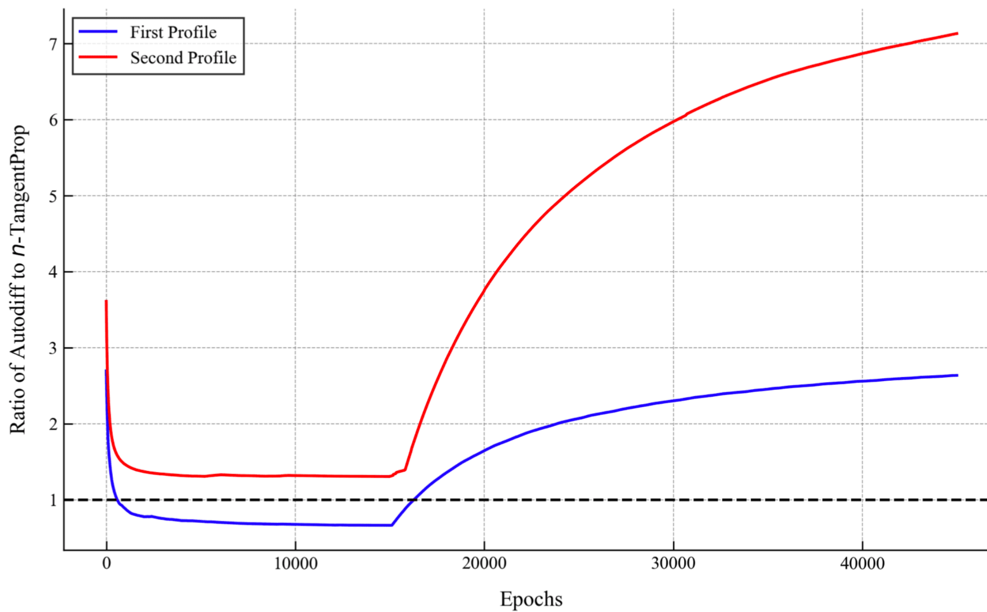


Figure 6. Results from training a PINN to find the first smooth profile for Equation 7. The model is trained for 15k epochs using the Adam optimizer and 30k epochs using L-BFGS. The top panel reports our training losses, the middle panel reports λ as a function of epochs, and the bottom panel shows the ratio in runtime between autodifferentiation and n-TANGENTPROP as a function of number of epochs. The bottom pannel shows that n-TANGENTPROP is 2.5x faster for end-to-end training than autodifferentiation. The horizontal line in the bottom panel indicates a runtime ratio of 1.

Figure 6 plots the ratio of execution times of autodifferentiation over n-TANGENTPROP and shows that we obtain significant speedups in computation time using n-TANGENTPROP instead of autodifferentiation. We were only able to compute the timing comparison between autodifferentiation and n-TANGENTPROP for the first two profiles since the computational time for the third profile using autodifferentiation exceeded our allowable computation time of 24 hours. For the first profile, which requires taking three derivatives, we obtain an end-to-end speed up of over 2.5 times. For the second profile, which requires taking five derivatives, we obtain an end-to-end speed up of over 7 times. We were able to compute the third profile, which requires taking seven derivatives, in a little under 1 hour using n-TANGENTPROP , and the projected time for auto-differentiation was over 25 hours, giving an expected speed-up of at least 25 times.

Using n-TANGENTPROP we were also able to compute the fourth profile, which requires taking nine derivatives. Using n-TANGENTPROP we were able to run the 45k epochs in a little under an hour and a half. We discuss our results further in Appendix -A, which we think are interesting in their own right. We stress that computing this fourth profile is untenable using autodifferentiation, as the time and space complexity render attempts at computation impossible. We estimate that computing the fourth profile using autodifferentiation would take at least 100 hours (about four days).

Observe from Figure 6 that the most dramatic improvements come when we switch to the L-BFGS optimizer, which uses multiple forward passes to perform a line search. Because n-TANGENTPROP has more favorable forward pass dynamics (c.f. Figures 2 and 3), the performance improvements are much more pronounced during L-BFGS optimization. This emphasizes the advantage afforded by n-TANGENTPROP : to obtain high-accuracy results we often use L-BFGS and Sobolev loss (see Equation 2). These two accuracy improvements require taking higher-order derivatives more frequently, which are the two areas that n-TANGENTPROP shows the best improvement in. Thus, for the high-accuracy training phase for PINNs, n-TANGENTPROP yields significant performance improvements.

We suspect that the dip below a ratio of 1 that we see in Figure 6 for computing the first profile can be mitigated through further optimizations of our implementation, and we hypothesize once again that the dip is likely due to efficiencies afforded by graph pruning and operator fusing in PyTorch.

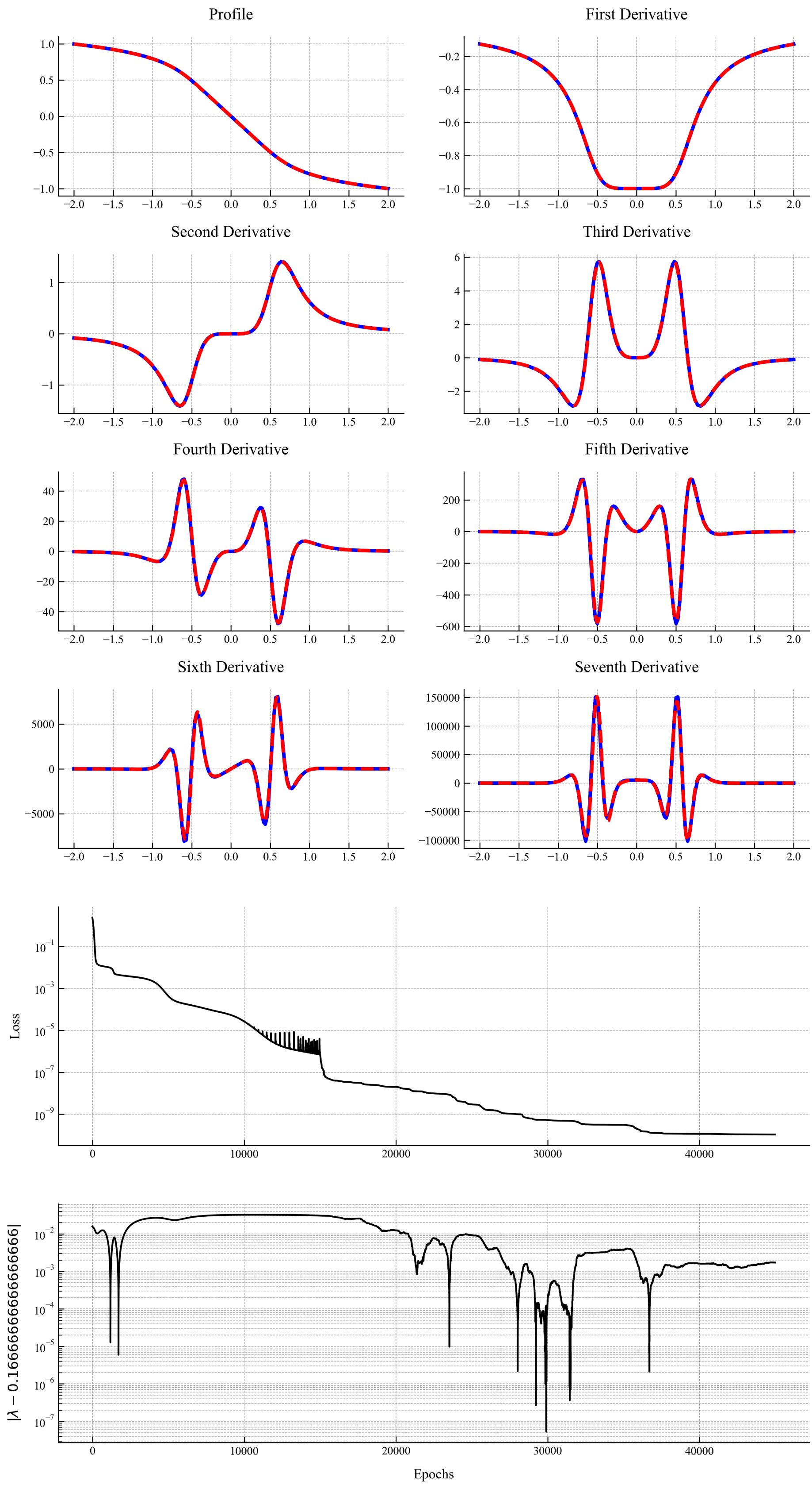

Profile
First Derivative
Second Derivative
Third Derivative
Fourth Derivative
Fifth Derivative
Sixth Derivative
Seventh Derivative
Loss
|λ − 0.1666666666666666|
Epochs

Figure 7. Results from training a PINN to solve (7) with λ constrained to the range $[1/7, 1/5]$. The only smooth solution to (7) contained in this parameter range corresponds to $\lambda = 1/6$. The first four rows show our learned solution (dashed red) and its derivatives compared to the true solution (solid blue). The second to last row shows the PINN training loss as a function of epochs. The model was trained for 15k epochs using the Adam optimizer and 30k epochs using the L-BFGS optimizer. The bottom row shows the inferred value for the parameter λ as a function of epochs. The bottom two rows are plotted with a logarithmic y-axis.

Figure 7 shows the result of training a PINN to find the third smooth profile of (7). This is the first time that we are aware of that this profile has been numerically computed with a floating value of λ using either a PINN or a traditional solver. We show the learned solution with a dashed red line superimposed over the true solution in blue. This profile is already computationally expensive to compute using autodifferentation and n-TANGENTPROP opens the door to performing novel studies on equations requiring a high-number of derivatives.

V. Conclusion

We introduced the n-TANGENTPROP formalism and demonstrated both theoretically and empirically that implementing our formalism in the context of PINNs dramatically reduces training times. We showed that for derivative-intensive PINN applications like finding high-order solutions to self-similar equations, n-TANGENTPROP not only offers improvements in end-to-end training times but allows the computation of previously untenable solutions. Our results are a step in the direction of making PINNs a more competitive numerical method for difficult forward and inverse problems. We recommend that our formalism be adopted by PINN implementations going forward to ensure faster training of PINNs.

We hope that our work allows the PINN community to explore more complicated problems, deeper and wider network architectures, and allow for researchers who do not have access to powerful computer to participate in furthering PINN research.

In this paper we have not focused on the optimization of our algorithm. We think that with optimization choices like implementing the underlying logic in C++ instead of Python that the

performance gap between n-TANGENTPROP and autodifferentiation would widen even further.

Appendix A. Additional Details for the Self-Similar Burgers Experiments

We report the results from running the self-similar Burgers experiment to find the smooth stable and unstable profiles. We were not able to reproduce the accuracy reported in [25], however we think that our results are important in demonstrating that their proposed methodology is robust, at least in theory. Furthermore, we report several new observations that we think are relevant to the training dynamics for such a problem.

We train our network using a Sobolev loss function (see Equation 2 function with $m = 1$ and additionally add a high-order loss term

$$L^*(u_\theta) = \frac{1}{N^*} \sum_{k=1}^{N^*} \left| \partial_x^n R(u_\theta, x_k) \right|^2,$$

where R is the residual of the self-similar Burgers equation and the samples x_k are taken from a small subset of collocation points centered at the origin, not the entire training domain. Our implementation contains many more subtle details that we omit for the sake of brevity and we encourage the reader to download our code to see the full implementation.

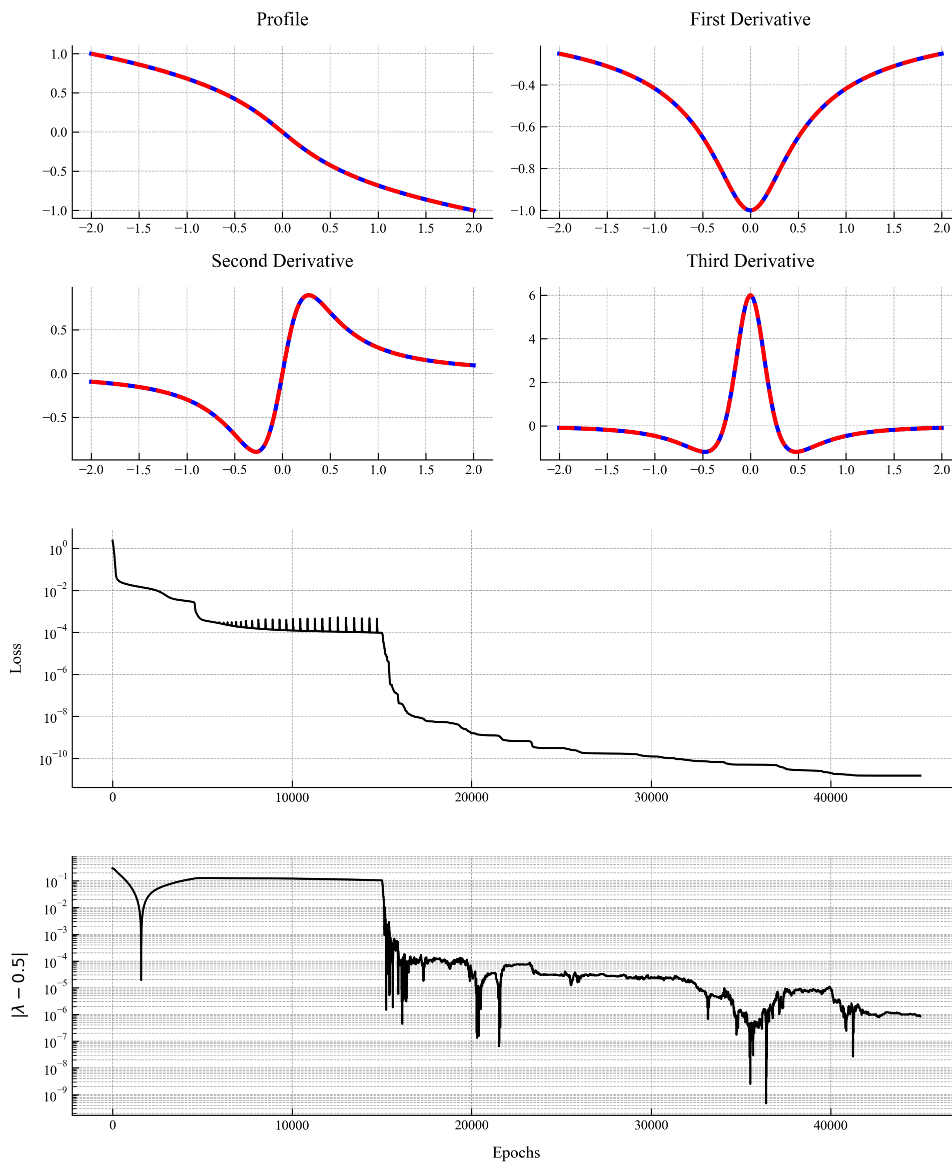


Figure 8. Results from training a PINN to solve (7) with λ constrained to the range $[1/3, 1]$. The only smooth solution to (7) contained in this parameter range corresponds to $\lambda = 1/2$. The first two rows show our learned solution (dashed red) and its derivatives compared to the true solution (solid blue). The second to last row shows the PINN training loss as a function of epochs. The model was trained for 15k epochs using the Adam optimizer and 30k epochs using the L-BFGS optimizer. The bottom row shows the inferred value for the parameter λ as a function of epochs. The bottom two rows are plotted with a logarithmic y-axis.

While we were not able to match the accuracy reported by [25], we were able to get our implementation to perform well in finding the first three smooth solutions to (7). However our method did not satisfactorily solve for the fourth profile. We discuss this in more detail below.

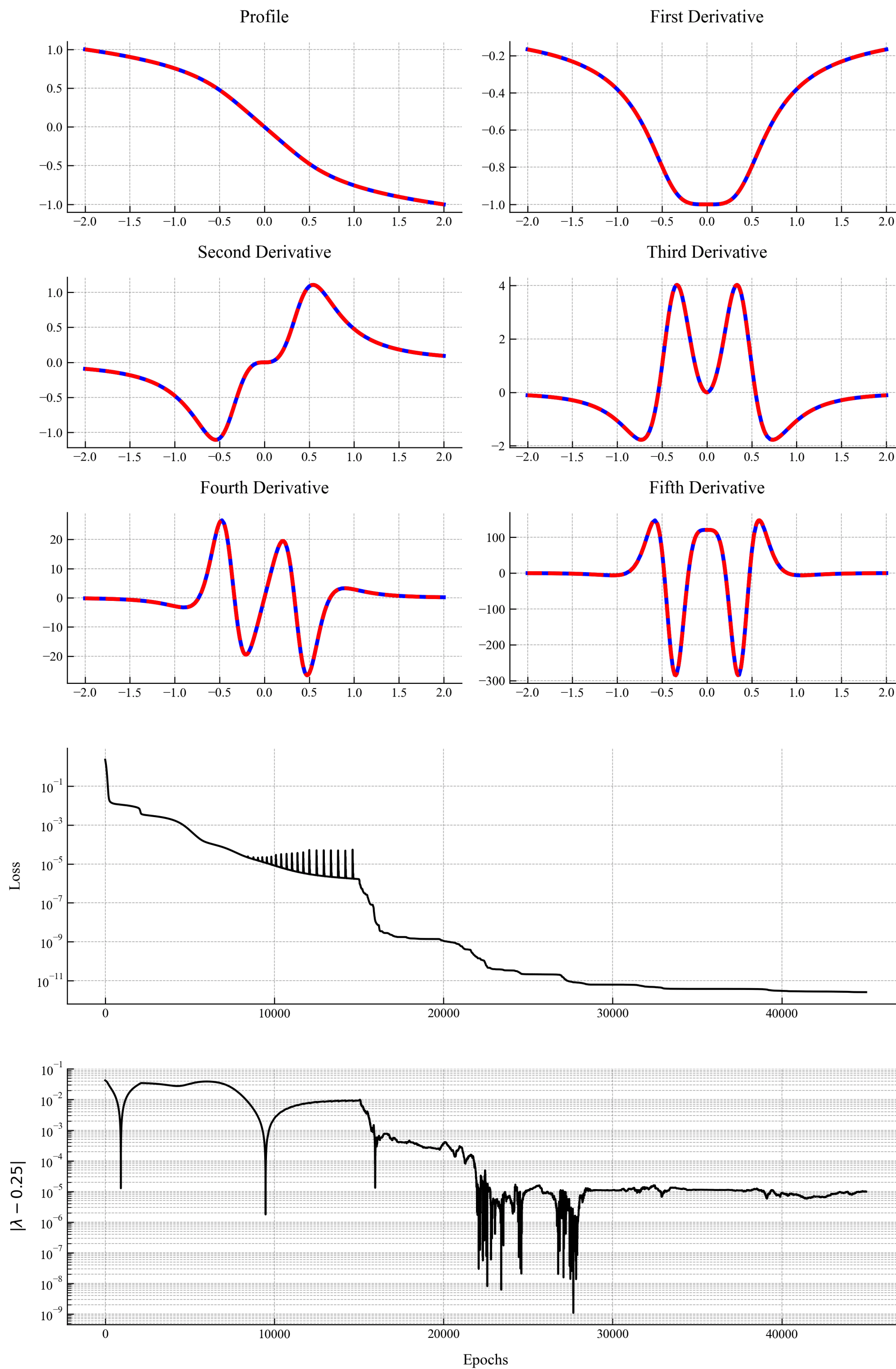
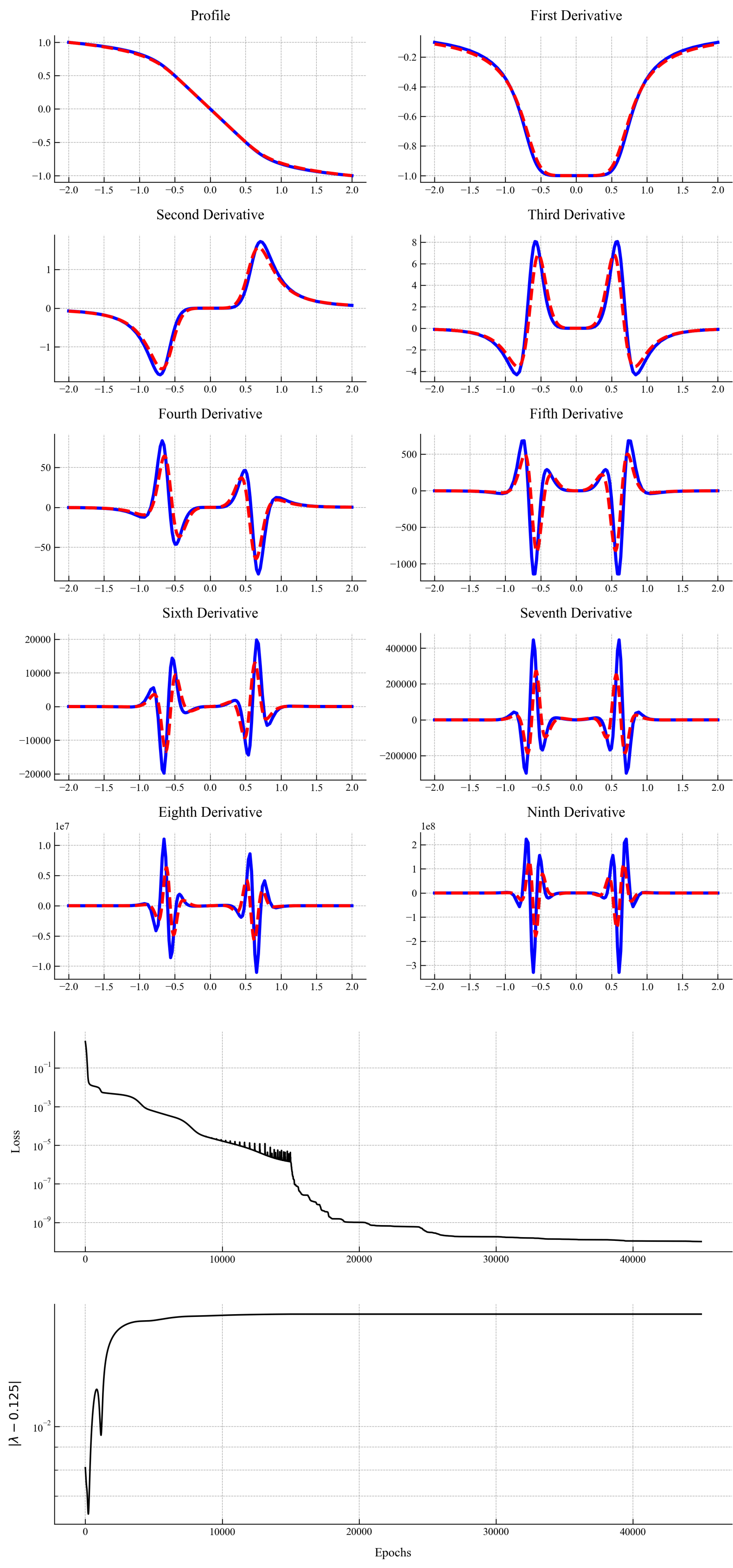


Figure 9. Results from training a PINN to solve (7) with λ constrained to the range $[1/5, 1/3]$. The only smooth solution to (7) contained in this parameter range corresponds to $\lambda = 1/4$. The first three rows show our learned solution (dashed red) and its derivatives compared to the true solution (solid blue). The second to last row shows the PINN training loss as a function of epochs. The model was trained for 15k epochs using the Adam

optimizer and 30k epochs using the L-BFGS optimizer. The bottom row shows the inferred value for the parameter λ as a function of epochs. The bottom two rows are plotted with a logarithmic y-axis.

For all of the profiles we found, we report our inferred value of λ as a function of training epochs. We think that this metric is an important scalar quantity and it's evolution is not reported in the original paper [25]. Of particular importance is the apparent inability of the Adam optimizer to satisfactorily converge to the correct value of λ. We see a sharp decrease in the error of λ once we begin to use the L-BFGS optimizer. We think that this phenomenon is interesting and may indicate that the first order derivatives of the residual with respect to the parameter λ is insufficient to capture the true dependency of the solution on λ. Understanding this dependency more deeply may lead to better training algorithms for these types of problems.

Notably, our code failed to adequately converge to the fourth profile corresponding to $\lambda = \frac{1}{8}$ (see Figure 10). Due to the nature of this work we did not pursue this point further and want to emphasize that we are not claiming that the methodology proposed in [25] cannot be applied to higher-order profiles. We hypothesize that the nature of the problem makes it more difficult for PINN or non-PINN solvers to find a solution. We are constraining the ninth derivative to be close to zero near the origin, but due to the relatively large magnitude of the ninth derivative, minor fluctuations in the network output will result in large changes to the ninth derivative. This alone may be enough to render our solver incapable of converging to the desired solution as we are using a fixed ratio to balance the relative terms in our loss function (see [12] for further discussion about why multi-target training is difficult). Put another way, we suspect that the loss function we are using does not properly account for the fact that we are taking nine derivatives and that as a consequence the parameter λ is not receiving good gradient signal.



 doi.org/10.32388/92OY7E

Figure 10. Results from training a PINN to solve (7) with λ constrained to the range $[1/9, 1/7]$. The only smooth solution to (7) contained in this parameter range corresponds to $\lambda = 1/8$. The first five rows show our learned solution (dashed red) and its derivatives compared to the true solution (solid blue). The second to last row shows the PINN training loss as a function of epochs. The model was trained for 15k epochs using the Adam optimizer and 30k epochs using the L-BFGS optimizer. The bottom row shows the inferred value for the parameter λ as a function of epochs. The bottom two rows are plotted with a logarithmic y-axis.

The purpose of this study was not to find the optimal hyperparameters for computing these higher-order profiles. Rather, the point of our study is to demonstrate that computing these higher-order profiles is feasible. We stress that using autodifferentitation to find this fourth profile would likely take several days on a state-of-the-art GPU, and we were able to compute it in less than two hours. We leave the refinement of model accuracy to future studies.

Acknowledgments

The author thanks Dr. Patrice Simard for stimulating discussions throughout the completion of this work. This work was partially completed while the author was employed by Hummingbird.ai.

Footnotes

[1] See for example https://discuss.pytorch.org/t/optimizer-with-line-search/19465 and https://discuss.pytorch.org/t/l-bfgs-b-and-line-search-methods-for-l-bfgs/674 for further discussion of this deficiency.

References

1. a, b, c, d *Simard P, Victorri B, LeCun Y, Denker J (1991). "Tangent prop-a formalism for specifying selected invariances in an adaptive network". Advances in Neural Information Processing Systems.* **4**.
2. a, b, c, d, e, f *Raissi M, Perdikaris P, Karniadakis GE (2019). "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". Journal of Computational Physics.* **378**: *686–707.*

doi.org/10.32388/92OY7E

3. ^McGreivy N, Hakim A (2024). "Weak baselines and reporting biases lead to overoptimism in machine learning for fluid-related partial differential equations". Nature Machine Intelligence. pp. 1–14.
4. a, b, cWang S, Sankaran S, Wang H, Perdikaris P (2023). "An expert's guide to training physics-informed neural networks". arXiv preprint arXiv:2308.08468. Available from: https://arxiv.org/abs/2308.08468.
5. a, bBaydin AG, Pearlmutter BA, Radul AA, Siskind JM (2018). "Automatic differentiation in machine learning: a survey". Journal of Machine Learning Research. 18: 1–43.
6. a, bAntonion K, Wang X, Raissi M, Joshie L (2024). "Machine Learning Through Physics--Informed Neural Networks: Progress and Challenges". Academic Journal of Science and Technology. 9 (1): 46–49.
7. a, bFarea A, Yli-Harja O, Emmert-Streib F (2024). "Understanding Physics-Informed Neural Networks: Techniques, Applications, Trends, and Challenges". AI. 5 (3): 1534–1557.
8. a, bWang H, Cao Y, Huang Z, Liu Y, Hu P, Luo X, Song Z, Zhao W, Liu J, Sun J, et al. Recent advances on machine learning for computational fluid dynamics: A survey. arXiv preprint arXiv:2408.12171. 2024.
9. a, bEvans LC. Partial differential equations. 19th ed. Providence (RI): American Mathematical Society; 2022.
10. ^Cybenko G (1989). "Approximation by superpositions of a sigmoidal function". Mathematics of Control, Signals and Systems. 2 (4): 303–314.
11. ^Hornik K (1991). "Approximation capabilities of multilayer feedforward networks". Neural networks. 4 (2): 251–257.
12. a, bBischof R, Kraus M (2021). "Multi-objective loss balancing for physics-informed deep learning". arXiv preprint arXiv:2110.09813. Available from: https://arxiv.org/abs/2110.09813.
13. ^Wang S, Teng Y, Perdikaris P (2020). "Understanding and mitigating gradient pathologies in physics-informed neural networks". arXiv preprint arXiv:2001.04536. Available from: https://arxiv.org/abs/2001.04536.
14. ^Yang Y, He J (2024). "Deeper or wider: A perspective from optimal generalization error with sobolev loss". arXiv preprint arXiv:2402.00152. Available from: https://arxiv.org/abs/2402.00152.
15. ^Krishnapriyan A, Gholami A, Zhe S, Kirby R, Mahoney MW (2021). "Characterizing possible failure modes in physics-informed neural networks". Advances in Neural Information Processing Systems. 34: 26548–26560.
16. ^Rathore P, Lei W, Frangella Z, Lu L, Udell M (2024). "Challenges in training PINNs: A loss landscape perspective". arXiv preprint arXiv:2402.01868. Available from: https://arxiv.org/abs/2402.01868.

17. ^Wang S, Yu X, Perdikaris P (2022). "When and why PINNs fail to train: A neural tangent kernel perspective". Journal of Computational Physics. **449**: 110768.

18. ^De Ryck T, Bonnet F, Mishra S, de B\'ezenac E (2023). "An operator preconditioning perspective on training in physics-informed machine learning". arXiv preprint arXiv:2310.05801. Available from: https://arxiv.org/abs/2310.05801.

19. ^Markidis S (2021). "The old and the new: Can physics-informed deep-learning replace traditional linear solvers?" Frontiers in big Data. **4**: 669097.

20. ^Rahaman N, Baratin A, Arpit D, Draxler F, Lin M, Hamprecht F, Bengio Y, Courville A (2019). "On the spectral bias of neural networks". In: International conference on machine learning. PMLR; 2019. p. 5301–5310.

21. ^Xu ZQJ, Zhang Y, Luo T, Xiao Y, Ma Z (2019). "Frequency principle: Fourier analysis sheds light on deep neural networks". arXiv preprint arXiv:1901.06523. Available from: https://arxiv.org/abs/1901.06523.

22. ^Czarnecki WM, Osindero S, Jaderberg M, Swirszcz G, Pascanu R (2017). "Sobolev training for neural networks". Advances in neural information processing systems. **30**.

23. a, bMaddu S, Sturm D, Müller CL, Sbalzarini IF (2022). "Inverse Dirichlet weighting enables reliable training of physics informed neural networks". Machine Learning: Science and Technology. **3** (1): 015026.

24. ^Son H, Jang JW, Han WJ, Hwang HJ (2021). "Sobolev training for physics informed neural networks". arXiv preprint arXiv:2101.08932. Available from: https://arxiv.org/abs/2101.08932.

25. a, b, c, d, e, f, g, h, i, j, k, l, mWang Y, Lai C-Y, G\u00f3mez-Serrano J, Buckmaster T (2023). "Asymptotic self-similar blow-up profile for three-dimensional axisymmetric Euler equations using neural networks". Physical Review Letters. **130** (24): 244002.

26. ^Chiu P-H, Wong JC, Ooi C, Dao MH, Ong Y-S (2022). "CAN-PINN: A fast physics-informed neural network based on coupled-automatic--numerical differentiation method". Computer Methods in Applied Mechanics and Engineering. **395**: 114909.

27. ^Sharma R, Shankar V (2022). "Accelerated training of physics-informed neural networks (pinns) using meshless discretizations". Advances in Neural Information Processing Systems. **35**: 1034–1046.

28. ^Jahani-Nasab M, Bijarchi MA (2024). "Enhancing convergence speed with feature enforcing physics-informed neural networks using boundary conditions as prior knowledge". Scientific Reports. **14** (1): 23836.

29. ^Penwarden M, Jagtap AD, Zhe S, Karniadakis GE, Kirby RM (2023). "A unified scalable framework for causal sweeping strategies for physics-informed neural networks (PINNs) and their temporal decomposi

tions". Journal of Computational Physics. ***493****: 112464.*

30. ^*Nabian MA, Gladstone RJ, Meidani H (2021). "Efficient training of physics-informed neural networks via importance sampling". Computer-Aided Civil and Infrastructure Engineering.* ***36*** *(8): 962–977.*
31. a, b, c*Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems. 2019;* ***32****.*
32. a, b*Roman S (1980). "The formula of Faa di Bruno". The American Mathematical Monthly.* ***87*** *(10): 805–809.*
33. ^*Brualdi RA. Introductory Combinatorics. 5th ed. Pearson; 2010.*
34. ^*Hardy GH, Ramanujan S (1917). "Asymptotic Formulae for the Distribution of Integers of Various Types". Proceedings of the London Mathematical Society.* ***2*** *(1): 112–132.*
35. ^*Shi HJM, Mudigere D (2020). PyTorch L-BFGS. Available from: https://github.com/hjmshi/PyTorch-LBFGS.*
36. ^*Nocedal J, Wright SJ. Numerical optimization. Springer; 1999.*
37. ^*Haberman R. Elementary applied partial differential equations: with Fourier series and boundary value problems. Prentice-Hall Englewood Cliffs, NJ; 1987.*
38. ^*Barenblatt GI, Zel'Dovich Ya B (1972). "Self-similar solutions as intermediate asymptotics". Annual Review of Fluid Mechanics.* ***4*** *(1): 285–312.*
39. a, b, c*Eggers J, Fontelos MA. Singularities: formation, structure, and propagation. Cambridge University Press; 2015. 53 p.*
40. ^*Chickering KR, Moreno-Vasquez RC, Pandya G (2023). "Asymptotically self-similar shock formation for 1D fractal Burgers' equation". SIAM Journal on Mathematical Analysis.* ***55*** *(6): 7328–7360.*
41. ^*Oh SJ, Pasqualotto F. Gradient blow-up for dispersive and dissipative perturbations of the Burgers equation. Archive for Rational Mechanics and Analysis.* ***248*** *(3): 54. 2024.*
42. ^*Biasi A. Self-similar solutions to the compressible Euler equations and their instabilities. Communications in Nonlinear Science and Numerical Simulation.* ***2021****;* ***103****: 106014.*

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: This work was partially completed while the author was employed by Hummingbird.ai.