Matrix-Free Trust-Region Newton-Conjugate-Gradient Optimizer in Deep Learning

Bryan Boyd, Jesen Tanadi

April 23, 2025

Summary

In this project we explore the use of Trust-Region Newton-Conjugate-Gradient (TRNCG) (specifically CG Steihaug) as an optimizer in training a neural network. Specifically, we leverage the forward and backpropagation process of the neural network—via reverse-mode automatic differentiation (AD)—to perform matrix-free computations of the action of the Hessian matrix on a vector. We compare the results of this approach with the results of a stochastic gradient descent optimizer (Adam) in approximating a variety of "elementary" mathematical functions (e.g., linear, quadratic, trigonometric), as well as in approximating a solution to the forward heat problem with the use of a Physics-Informed Neural Network (PINN). We observe that while TRNCG can converge to a high-quality approximation in significantly fewer iterations than Adam, each iteration may be more computationally intensive, resulting in slower overall training.

Background and Theory

Neural Networks

Mathematically, neural networks (NN) can be represented as compositions of affine transformations (parameterized by weights and biases) and a nonlinear activation function.[2] Specifically, we express our neural network as

$$\mathcal{N}(\mathbf{x};\sigma,\mathbf{w}_1,\ldots,\mathbf{w}_n)=\mathbf{w}_n^\intercal\,\sigma(\mathbf{w}_{n-1}^\intercal\,\sigma(\ldots\mathbf{w}_2^\intercal\,\sigma(\mathbf{w}_1^\intercal\mathbf{x})\ldots))), \tag{1}$$

where $\mathbf{x}$ is our input, $\{\mathbf{w}\}_i^n$ is the set of weights and biases, and σ is our activation function. The objective in training a neural network is to minimize the difference between the approximated value $\hat{\mathbf{y}}=\mathcal{N}(\mathbf{x})$ and the true value $\mathbf{y}$. For instance, with Mean Squared Error (MSE), we can denote the loss or objective function as

$$f(\mathbf{x},\mathbf{y})=\frac{1}{n}\left\|\mathcal{N}(\mathbf{x})-\mathbf{y}\right\|_2^2. \tag{2}$$

In the case of Physics-Informed Neural Networks, the total loss function is the sum of a series of loss functions, each associated with e.g., the differential equation itself, boundary conditions, initial conditions, and additional data. We express our PINN's loss function as the convex combination

$$f_{\text{total}}=\lambda(f_{\text{BC}}+f_{\text{IC}})+(1-\lambda)f_{\text{PDE}}, \tag{3}$$

where λ is a tuning parameter.

In general, the loss function of a neural network with more than one hidden layer is neither convex nor concave; instead, it can be visualized as a complex landscape containing many saddle points and local minima and maxima.[3, 7] This is illustrated in Figure 3a, where we see concave and convex regions—as well as saddle points—along the path traversed by the TRNCG optimizer. Despite this complexity, optimization algorithms are generally able to find suitable approximations; we explore this further in the Results and Discussion section.

This project utilizes a Fully-Connected Network (FCN) architecture to find approximations to the following elementary mathematical functions: linear, $y=5x+3$; quadratic, $y=\left(x-\frac{1}{2}\right)^2$; exponential, $y=e^{-x}$; and sinusoidal, $y=\sin(x-\pi)$, $y=\sin(5x)$, $y=\sin(10x)$; as well as the transient heat equation:

$$\begin{aligned} u_t-\nabla\cdot(\alpha\nabla u)&=0, && \text{in } (0,1)^2\times(0,5),\\ u&=0, && \text{along bottom and right boundaries}\\ u&=1, && \text{along top and left boundaries}\\ u|_{t=0}&=0. \end{aligned}$$

Variations in the frequency of the sin function are chosen to test the limits of both Adam and TRNCG, as neural networks are known to have difficulty learning periodic functions with high frequency.[14, 15]

Trust-Region Method

Like other numerical optimization techniques, Trust-Region (TR) methods approach the solution by iteratively minimizing the objective function. However, unlike methods such as stochastic gradient descent, TR methods construct a quadratic model of the objective around the current iterate $\mathbf{x}_k$, along with a region within which this model is considered a sufficiently accurate approximation of the objective function. We denote this model

$$m_k(\mathbf{p}) = f_k + g_k^\intercal \mathbf{p} + \frac{1}{2}\mathbf{p}^\intercal B_k \mathbf{p}, \tag{4}$$

where $\mathbf{p}$ is a step, $f_k = f(\mathbf{x}_k)$, $g_k = \nabla f_k$, and B_k is a symmetric matrix. In this work, we assume $f_k \in C^2$ and take $B_k = \nabla^2 f_k$ (i.e., Newton's method). Then by the symmetry of second derivatives (Schwarz's theorem), $B_k^\intercal = B_k$. The computation of $B_k\mathbf{p}$ is elaborated in the Computing the Action of the Hessian subsection.

To obtain the (approximately) optimum step $\mathbf{p}^*$ for some trust-region of radius $\Delta_k > 0$ and for m_k defined in Equation (4), we solve the subproblem

$$\begin{aligned} \mathbf{p}^* = \arg\min \quad & m_k(\mathbf{p}) \\ \text{subject to} \quad & \|\mathbf{p}\| \leq \Delta_k. \end{aligned} \tag{5}$$

In this project, we rely on CG-Steihaug to find an approximation to Equation (5). This is discussed in more detail in the next subsection.

Having found $\mathbf{p}_k = \mathbf{p}^*$, our next task is to choose a radius of the trust-region for the next iteration Δ_{k+1}. To do so, we compute the ratio

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{p}_k)} = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{f(\mathbf{x}_k) - m_k(\mathbf{p}_k)} = \frac{\text{actual reduction of objective}}{\text{predicted reduction of objective}} \tag{6}$$

with the idea that if ρ_k is either negative or close to 0 then the algorithm is not deceasing the objective as much as predicted and the trust-region needs to be shrunk—typically to a fraction of $\|\mathbf{p}_k\|$. On the other hand, if our current step is at the boundary of the trust-region and ρ_k is close to 1, then the model is an appropriate approximation of the objective and we can enlarge the trust-region. On a practical level, the TR algorithm we implement closely follows *Algorithm 4.1 (Trust-Region)* as described in [9].

Newton-Conjugate-Gradient Method

Many optimization problems, including training a neural network, involve thousands to millions of variables. As shown in the previous section, a main component of the trust-region method involves solving the quadratic subproblem. In order to solve this directly, the computation of the Hessian matrix is required, which is extremely computationally-intensive with a large number of variables. This is because computing the Hessian matrix explicitly requires $O(n^2)$ operations. To deal with this issue, a variation of the Conjugate-Gradient method, the Newton-Conjugate-Gradient method is introduced. The main idea behind this method is to approximately solve the Trust-Region subproblem iteratively without explicitly calculating or storing the Hessian matrix, rather this method relies on computing Hessian-vector products efficiently via automatic differentiation. The computation of these Hessian-vector products reduce the complexity to $O(n)$.

This method begins by setting the initial residual r equal to the gradient g_k, and setting the initial search direction $\mathbf{d}$ equal to the negative residual $(-g_k)$. During each iteration, there are two main conditions that determine when to stop iterating—negative curvature detection and trust-region boundary detection. In the first case, if there is a negative curvature detected along the current search direction (i.e. $\mathbf{d}^\intercal B_k \mathbf{d} \leq 0$), this indicates that the quadratic approximation isn't convex in the search direction. Thus, the algorithm determines the step length that exactly meets the trust-region boundary and terminates. Neural networks are often non-convex, making negative curvature common, and therefore this negative curvature detection is very practical for these problems. For the second case, if the calculated full step for the current iteration would surpass the trust-region radius, the algorithm calculates the step length that is necessary to exactly reach the boundary of the trust-region, and terminates there. When neither of these stopping criteria is triggered, the CG-Steihaug method proceeds by performing standard conjugate-gradient updates on step length, residual, and search direction. The iterations continue until the residual is sufficiently small, and thus the approximate solution is found to the trust-region subproblem. The CG-Steihaug algorithm that we implement, and which is described in this section, follows *Algorithm 7.2 (CG-Steihaug)* as described in [9].

Computing the Action of the Hessian

In TRNCG, the Hessian of the loss function $\nabla^2 f = B$ appears in multiple places—both in the TR component (i.e., in Equation (4)) and in the CG-Steihaug method. However, in all cases, we only need to compute its action on a vector rather than form the full matrix explicitly. To perform this operation, we rely on automatic differentiation (AD). Specifically, we use PyTorch's `autograd.functional.vhp()`, which applies reverse-mode AD twice in succession. We note that reverse-mode AD requires a potentially-large computational graph to be built and traversed, and this cost may outweigh the benefits gained from not generating the full Hessian. Further background on reverse-mode AD (used by PyTorch) can be found in [9].

Implementation Details

Networks and Data

The FCN used to approximate elementary functions—denoted $\mathcal{N}_e$—takes in scalar-valued inputs x and outputs a scalar value $y(x)$. It is comprised of 10 hidden layers with 10 neurons each, resulting in a total of 1,131 trainable parameters, including weights and biases. In contrast, the FCN for the heat equation—denoted $\mathcal{N}_p$—takes in vector-valued inputs $\mathbf{x} = [x, y, t]^\intercal$ (2-D location and time) and outputs the temperature $u(x, y, t)$. It comprises four hidden layers with 24 neurons each, resulting in a total of 2,521 trainable parameters. Both networks employ $\sigma = \tanh$ as their activation function; training and testing are implemented in PyTorch.[10]

Lastly, we discuss the training and testing data for both FCNs, $\mathcal{N}_e$ and $\mathcal{N}_p$. For all of elementary functions, $\mathcal{N}_e$ is trained with 10,000 points randomly sampled from a uniform distribution; i.e., $x \sim \mathcal{U}(-1.1, 1.1)$. The network is then tested with 10,000 points linearly sampled from the interval $[-1, 1]$. $\mathcal{N}_p$ is trained on 10,000 points generated by Latin hypercube sampling (LHS) within the space-time domain $[0, 1]^2 \times [0, 5] \subset \mathbb{R}^2 \times \mathbb{R}$, with 20% of the points sampled from the boundary to enforce the Dirichlet condition of the PDE. The network is then tested on 100,000 points sampled using LHS from the spatial domain $[0, 1]^2 \subset \mathbb{R}^2$ at $t = 1$.

Optimizer & Training Parameters

For both $\mathcal{N}_e$ and $\mathcal{N}_p$, we use an Adam learning rate of 10^{-3}. While not equivalent, we can think of this learning rate as analogous to the base step size of a typical first-order optimizer.[6]

Trust-Region Newton CG requires a few more hyper parameters. For all applications of TRNCG, we choose the following values:

- $\Delta_0 = 2$: The initial trust-region radius. This is chosen arbitrarily, but kept reasonably small.
- $\hat{\Delta} = 10$: The maximum trust-region radius. This is chosen to be fairly large so that the optimizer has the ability to escape saddle points.
- $\eta = 10^{-3}$: The acceptance threshold of the current iterate. That is, if the current $\rho_k > \eta$, we accept the step $\mathbf{p}_k$ found by the CG subproblem; otherwise, we reject it and find a new step at the next iteration. This is chosen to favor acceptance, so we do not stagnate at the same point for many iterations.
- $\boldsymbol{\varepsilon}_k = 10^{-8}$: The CG-Steihaug tolerance that allows the algorithm to return an inexact Newton solution at the current iterate. We choose $\boldsymbol{\varepsilon}_k$ to be very small to obtain accurate solutions.
- $j_{\max} = 500$: The maximum number of iterations for CG-Steihaug. We choose a large number so that the iteration terminates by one of the thresholding conditions instead. In practice, CG-Steihaug performs at most 27 iterations, as seen in Figure 3c.

For both optimizers, $\mathcal{N}_e$ is trained until its loss $f(\mathbf{x}, \mathbf{y}) \leq 10^{-6}$. We also impose a maximum number of iterations, set at 25,001, to ensure that training time remains manageable. On the other hand, $\mathcal{N}_p$ does not have a loss threshold, but has a maximum number of iterations set at 15,001.

Hessian-Vector Product vs. Vector-Hessian Product

We discuss our choice of PyTorch function for computing the action of the Hessian on a vector, $B\mathbf{v}$. As shown in the Trust-Region Method subsection, $f \in C^2 \Rightarrow B^\intercal = B$. From this, it follows that

$$B\mathbf{v} = B^\intercal \mathbf{v} = (\mathbf{v}^\intercal B)^\intercal. \tag{7}$$

Additionally, in PyTorch vectors are stored as 1-D `Tensor` objects without an explicit orientation. Thus, for $\mathbf{v} \in \mathbb{R}^n$ and $B \in \mathbb{R}^{n \times n}$,

$$\mathbf{v} = \mathbf{v}^\intercal \qquad \text{and} \qquad \mathbf{v}^\intercal B = B\mathbf{v}, \tag{8}$$

where both objects are treated as vectors in $\mathbb{R}^n$, regardless of shape notation.

Equations (7) and (8) indicate that, for this project, computing the vector–Hessian product using PyTorch's built-in `autograd.functional.vhp()` yields the same result as computing the Hessian–vector product with `hvp()`. As noted in the PyTorch documentation, `vhp()` is significantly more efficient than `hvp()`.[12, 13] Accordingly, we use `vhp()` in our implementation to benefit from its efficiency without compromising correctness.

This efficiency gain is a result of how PyTorch structures its automatic differentiation (AD) system. By default, PyTorch uses reverse-mode AD, which aligns naturally with the structure of forward and backward propagation in neural networks.[1] An inspection of the source code reveals that `vhp()` internally performs two successive applications of reverse-mode AD to compute the desired quantity.[11]

Results and Discussion

Elementary Functions

For all six elementary functions, both Adam and TR Newton-CG converge to the appropriate function as shown in Figure 1. Additionally, in Figure 4a we see that for all of them, the TRNCG optimizer converges in much fewer iterations than Adam. For many of these functions, we see that Adam reaches the maximum number of iterations allowed, and their losses do not go below the predefined threshold. In contrast, TR Newton-CG is able to converge rather quickly by using the additional information provided by the Hessian—i.e., the curvature of the loss landscape. For functions that are more complicated—such as the sinusoidal functions—the additional information of curvature still helps TRNCG converge in less iterations than Adam.

However, when we compare training time, as shown in Figure 4b, we see that as the complexity of the functions increase, the time it takes to train TR Newton-CG compared to Adam increases significantly. This can be particularly seen for the sinusoidal functions as well as the heat equation, as TR Newton-CG is more performant on the other three elementary functions. Even though Adam still takes more iterations, these are much cheaper and faster, and thus take significantly less time in general compared to TRNCG. Finally, with the heat equation we can see something similar to the high frequency sinusoidal functions, where non-convexity in the solution results in a higher training time for TR Newton-CG.

Figure 2 compares the change of the loss values optimized with TRNCG against those optimized with Adam for the first 500 iterations. We see that TR Newton-CG is able to significantly minimize the loss function early on, and often attains a lower loss value in a smaller number of iterations than Adam does for a longer training period. As an example, we specifically discuss the plot of the loss for $y = e^{-x}$. As can be seen, for the first few iterations of the TR Newton-CG method, there are a few spikes in the loss before it quickly drops down to a near zero loss. This likely indicates that our step $\mathbf{p}$ reaches the trust-region boundary, requiring an adjustment to the trust-region. Once the curvature of the neural network landscape is approximated by the trust-region, the loss rapidly declines as it begins to take near-optimal steps. In contrast, we observe that Adam starts at a smaller loss and gradually decreases to a near zero loss over more iterations. This is due to the fact that it is a first order optimizer with a small predefined learning rate, and thus each step it takes is typically small, resulting in more iterations to achieve optimality.

Heat Equation with PINN

In training our PINN, we observe that the TRNCG optimizer achieves a significantly smaller loss than the Adam optimizer; this is demonstrated in Figure 5. However, as shown in Figure 6, the network trained with TRNCG doesn't produce the correct solution in the allocated number of iterations.

We explored a number of potential explanations for this result, but did not reach a definitive conclusion. Some ideas we discussed are: ill-conditioned Hessian; incorrectly-sized TR (we observed that in our PINN training, CG-Steihaug only performed one iteration for every TR iteration, potentially exiting when $\|z\| \geq \Delta_k$); or a poorly-chosen PINN hyperparameter λ in Equation (3).

Convergence of Trust-Region Newton-Conjugate-Gradient

In trust-region methods, for $\eta > 0$ we are guaranteed convergence to a stationary point, which may or may not be a minimum; this is shown in *Theorem 4.2* in [9]. As mentioned in the Neural Networks subsection, the NN loss landscape is complex and highly non-convex, with many local minima and saddle points. In most cases, training difficulties in high dimensions often stem from the prevalence of saddle points.[4] This is a condition that our current technique doesn't address; in TRNCG, convergence to a solution is only guaranteed for positive definite Hessians.

In most practical cases, though, neural networks tend to reach a local minimum. Kawaguchi proved in [5] that in neural networks with linear activations, all local minima are equivalent to the global minimum. This result can be extended to networks with nonlinear activations and certain structures as shown in [8]. Further, [3] argues that even if the neural network's local minima are not equivalent to the global minimum, the loss landscape still contains many "high quality" local minima, in the sense that their loss values are close to that of the global minimum. These findings suggest that even if TRNCG converges to a local minimum, the resulting solution is often sufficiently accurate in practice. Moreover, as illustrated in Figure 3d, the successive shrink–expand cycles of the trust region radius aligns well with the expected local curvature of the loss landscape, confirming that the algorithm is dynamically adapting its step sizes as intended.

CG-Steinhaug inherits standard CG convergence and thus converges at least linearly towards a stationary point. Provided that we do not stagnate on a saddle point, each iteration of CG-Steihaug reduces the model m_k by generating steps that are at least as good as the Cauchy point, thereby guaranteeing global convergence. Further, assuming the Hessian is positive definite, it is shown by *Theorem 5.1* in [9] that any conjugate-gradient method converges to a solution $\mathbf{p}^* \in \mathbb{R}^n$ in n iterations. Finally, *Theorem 4.1* in [9] proves that a solution $\mathbf{p}^*$ to the TR subproblem satisfies the KKT conditions. In our case, since CG-Steihaug computes an approximate solution to the TR subproblem, we approximately satisfy the KKT conditions.

Figures

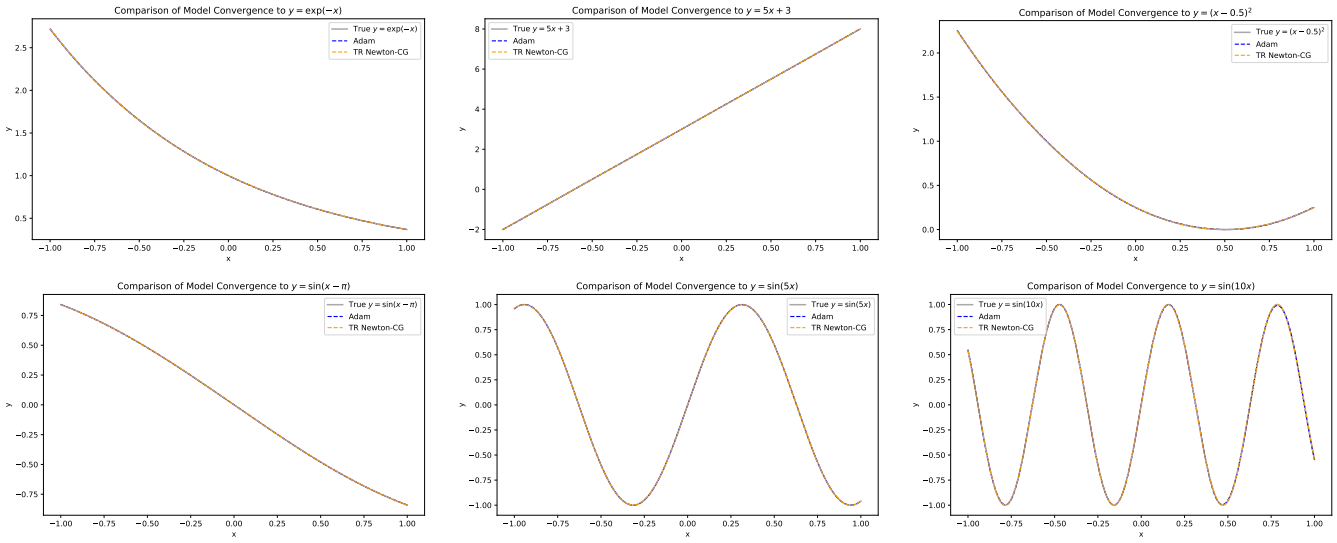


Figure 1: Model convergence to elementary functions.

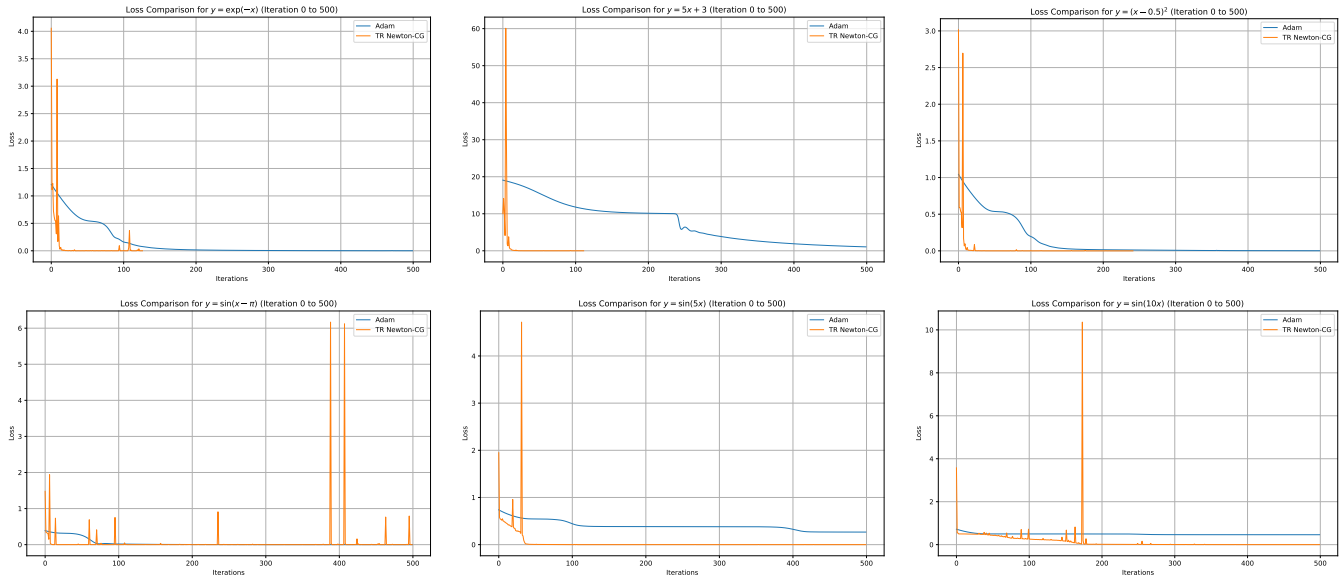
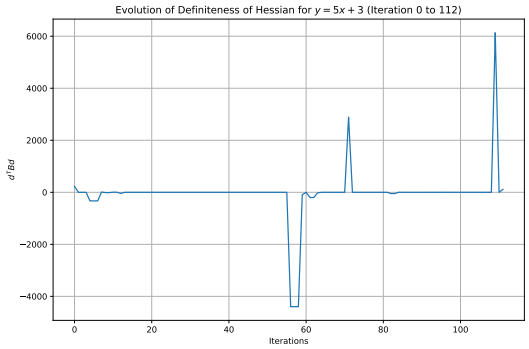
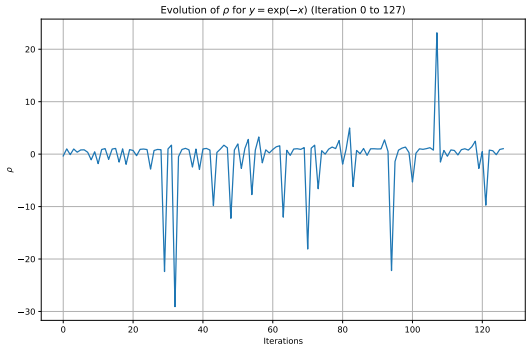


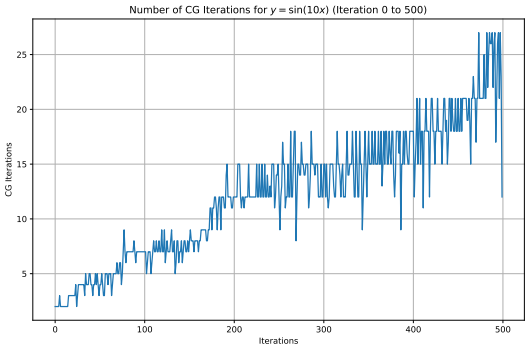
Figure 2: Comparison of loss values for the first 500 iterations of training $\mathcal{N}_e$. Across the top row, we observe that TRNCG reaches its loss threshold in less than 500 iterations.



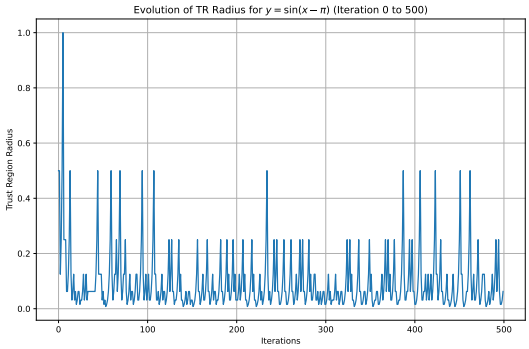
(a) Evolution of the definiteness of the Hessian of our loss function for $y = 5x + 3$. Even for a simple linear y, the loss landscape traversed by the optimizer contains concave and convex regions.



(b) Evolution of ρ_k per iteration for $y = e^{-x}$. Negative values of ρ_k indicates that the quadratic model m_k is not a good approximation; i.e., $f(\mathbf{x}_k + \mathbf{p}_k) > f(\mathbf{x})$.

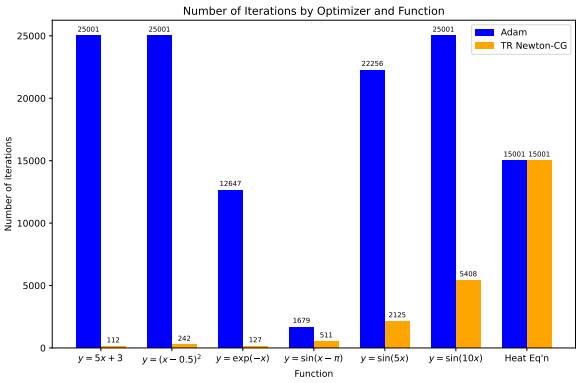


(c) Number of CG-Steihaug iterations per TR iteration for $y = \sin(10x)$, showing the highest number of CG iterations for all functions trained.

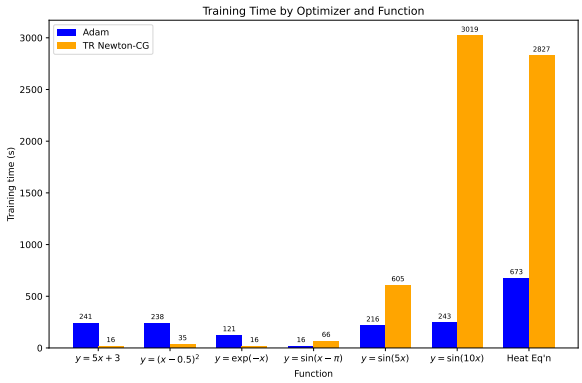


(d) Evolution of TR radius Δ_k for iterations 0 to 500 for $y = \sin(x - \pi)$. We see that the trust-region undergoes constant adjustments as we traverse through the loss landscape.

Figure 3: Summary figures showing some aspects of the TRCNG optimizer across different elementary functions.

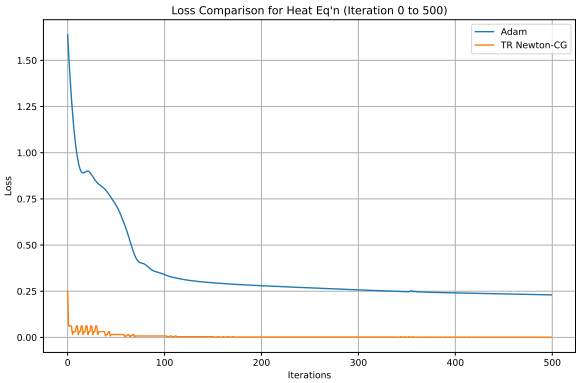


(a) Comparison of number of iterations to minimize each loss function up to a specified threshold value.



(b) Comparison of actual training time to minimize each loss function up to a specified threshold value.

Figure 4: Performance comparison between Adam and TRCNG.



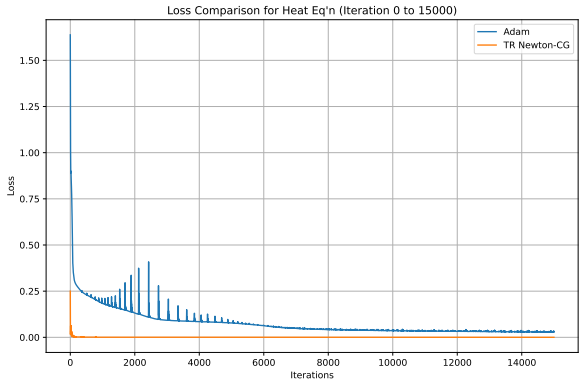


Figure 5: Comparison of loss values during $\mathcal{N}_p$ training, showing that TRNCG obtains lower loss values for the entire training period.

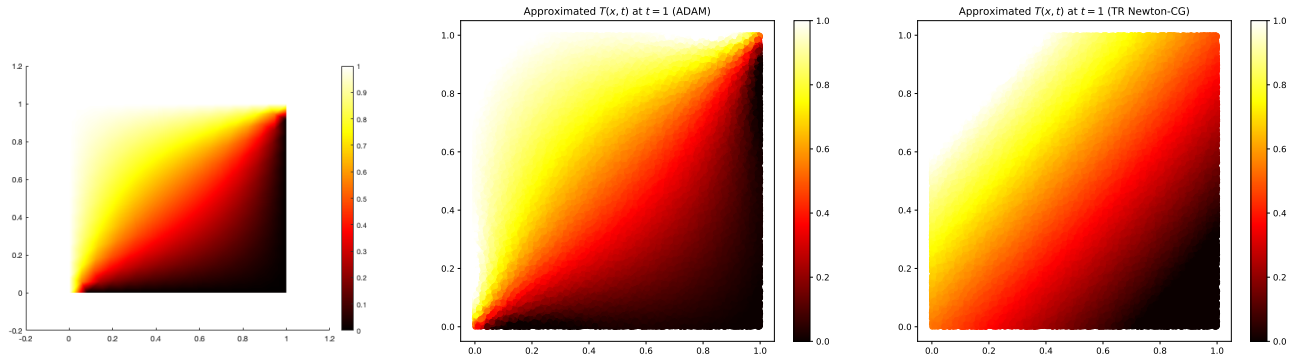


Figure 6: FEM solution to the heat equation (left) compared to $\mathcal{N}_p$ solution with Adam (middle) and with TRNCG (right).

Concluding Remarks

As we saw in the training examples, while computing the Hessian-vector product using AD is more efficient than directly computing the Hessian, it is still a costly operation that causes TRNCG to take a significant amount of time in optimizing large-scale problems. As reverse-mode AD requires a potentially large computational graph to be

built and traversed, the benefits of not generating a Hessian matrix are potentially negated. Further, in employing a second-order method, we must also contend with the possibility of ill-conditioned Hessians.

While neural networks equipped with TRNCG may converge in fewer iterations, it can also take a much longer time to train. On the other hand, first-order stochastic gradient descent optimizers such as Adam can take a comparable amount of time, even in the case where it requires more iterations to converge.

In addition to these performance aspects, TR Newton-CG is sensitive to its numerous hyper parameters, which includes the initial TR radius, as well as TR and CG tolerances, whereas for many applications of Adam, the only required parameter to be tuned is its learning rate. This, in addition to the performance considerations mentioned above, potentially contribute to Adam's popularity for general use cases.

References

[1] Atilim Gunes Baydin et al. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] Anna Choromanska et al. "The Loss Surfaces of Multilayer Networks". In: *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics* (2015).

[4] Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014.

[5] Kenji Kawaguchi. "Deep Learning without Poor Local Minima". In: *30th Conference on Neural Information Processing Systems* (2016).

[6] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. 2015.

[7] Hao Li et al. "Visualizing the Loss Landscape of Neural Nets". In: *32nd Conference on Neural Information Processing Systems* (2018).

[8] Quynh Nguyen and Matthias Hein. "The loss surface of deep and wide neural networks". In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 2017. arXiv: 1704.08045 [cs.LG]. URL: https://arxiv.org/abs/1704.08045.

[9] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Second. Springer, 2006.

[10] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[11] PyTorch Contributors. *PyTorch: `autograd.functional.vhp()` Source Code*. https://github.com/pytorch/pytorch/blob/v2.6.0/torch/autograd/functional.py\#L970. Accessed: 2025-04-18. 2024.

[12] PyTorch Developers. *torch.autograd.functional.hvp — PyTorch Documentation*. https://pytorch.org/docs/stable/generated/torch.autograd.functional.hvp.html. Accessed: 2025-04-18. 2024.

[13] PyTorch Developers. *torch.autograd.functional.vhp — PyTorch Documentation*. https://pytorch.org/docs/stable/generated/torch.autograd.functional.vhp.html. Accessed: 2025-04-18. 2024.

[14] Shijun Zhang et al. "Why Shallow Networks Struggle with Approximating and Learning High Frequency: A Numerical Study". In: (2023).

[15] Liu Ziyin, Tilman Hartwig, and Masahito Ueda. "Neural Networks Fail to Learn Periodic Functions and How to Fix It". In: *34th Conference on Neural Information Processing Systems* (2020).