Neural Network Optimization Under PDE Constraints

G. Eli Jergensen

12 Sep. 2019

Outline

- Motivation
- Constrained Optimization
  - Theory
  - Experimental Design
  - Results
- Nonlinear Projection
  - Theory
  - Experimental Design
  - Results
- Conclusion and Future Work

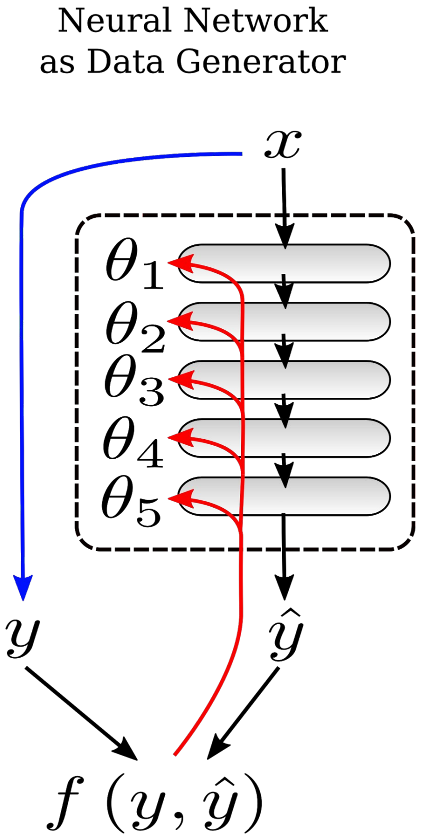
Motivation

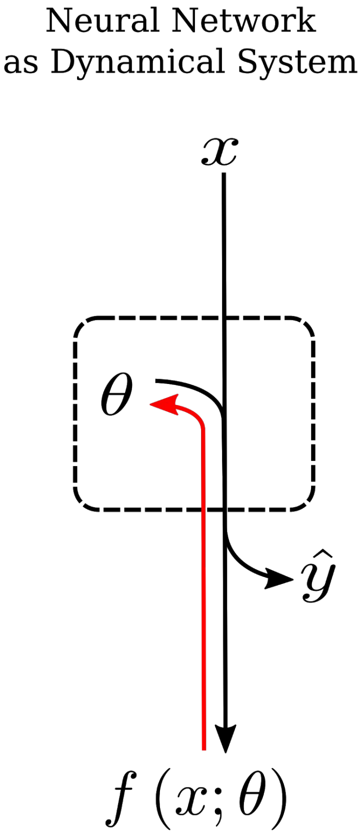
- Neural Networks are really powerful data generators
  - *e.g.* faces, cats, dogs
- Can be used to generate scientific data
  - *e.g.* weather forecasts, model fluids
- Currently don’t know or respect physical laws
  - *e.g. Conservation of energy, conservation of momentum, etc.*

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. arXiv preprint arXiv:1812.04948, 2018.

Neural Networks

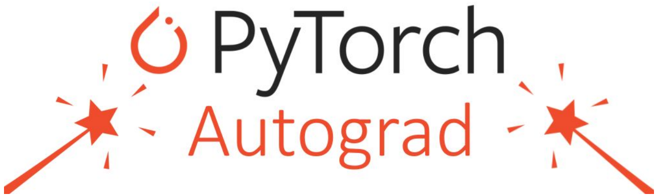
- Can model arbitrary functions
  $$x \mapsto y$$
- Parameterized by weights, θ
- Trained by use of a loss function, f
- Uses "truth" data and backpropagation to update weights





Partial Differential Equations and Neural Networks

- Most physical laws/relationships can be written as a PDE
- Neural networks themselves model functions
- Therefore, we can apply a PDE to a neural network as long as we can take derivatives
- Auto-differentiation does just that!
  - Idea: define a derivative for all simple operations. Derivatives for complex operations can be defined inductively using the chain rule



https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95

Methods for Constraining Neural Networks

1. Domain Specific
    a. *e.g.* If you want your model's outputs to have zero divergence, take the curl
2. Soft-Constraints / Regularization
    a. Simply add some extra terms to your loss function to handle constraint
    b. Pro: very computationally cheap (a couple extra additions)
    c. Con: doesn't guarantee the constraints are satisfied
3. Constrained Optimization
    a. Modify neural network training to be a constrained optimization problem
4. Nonlinear projection
    a. Take the parameters of the network and "project" them to the nearest point which satisfies a constraint

Methods for Constraining Neural Networks

1. Domain Specific
   a. *e.g.* If you want your model's outputs to have zero divergence, take the curl
2. Soft-Constraints / Regularization
   a. Simply add some extra terms to your loss function to handle constraint
   b. Pro: very computationally cheap (a couple extra additions)
   c. Con: doesn't guarantee the constraints are satisfied
3. Constrained Optimization
   a. Modify neural network training to be a constrained optimization problem
4. Nonlinear projection
   a. Take the parameters of the network and "project" them to the nearest point which satisfies a constraint

Constrained Optimization

Neural Network Training as an Optimization Problem

- Typical neural network training is a minimization problem:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \underset{x_B}{\mathbb{E}} [f(x_B; \theta)]$$

- Θ is the space of possible parameters, θ
- θ^* is the optimal parameters of the neural network
- x_B is a batch of inputs to the network
- f is the minimization (loss) function, which can be phrased as a function of the input batch and the current network parameters
- $\underset{x_B}{\mathbb{E}}$ indicates that we take the expectation over the batches of inputs

NN Training as a Constrained Optimization Problem

- We can upgrade the optimization problem to a constrained one by adding

$$\theta^* = \underset{\substack{\theta \in \Theta \\ \boldsymbol{g}(\theta)=\boldsymbol{0}}}{\operatorname{argmin}} \ \underset{x_B}{\mathbb{E}} \left[f(x_B; \theta) \right]$$

- $\boldsymbol{g}$ is an equality constraint function which is satisfied when

$$\boldsymbol{g}(\theta)=\boldsymbol{0}$$

- The constraint function implicitly defines a ***manifold*** of valid parameters*:

Constraint manifold: $\boldsymbol{g}^{-1}(\boldsymbol{0})$

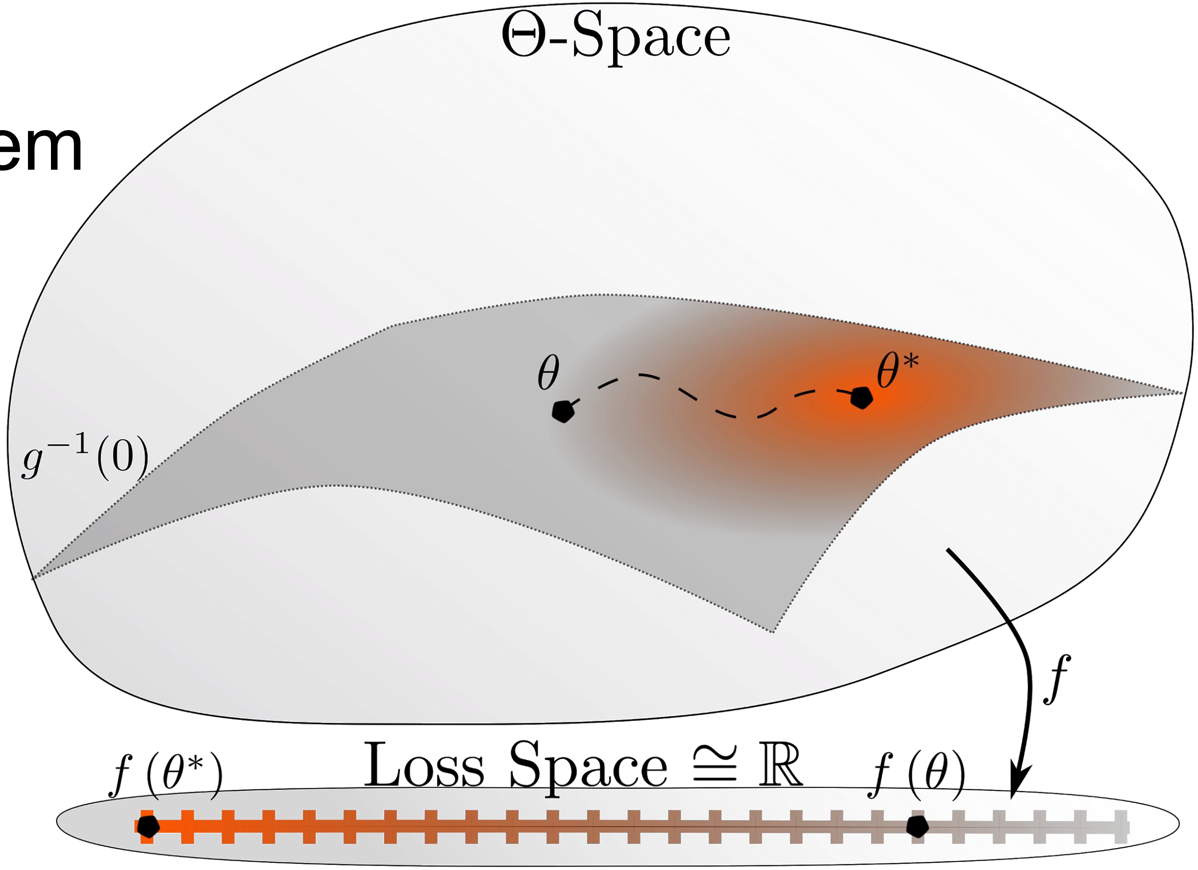
---

* Under the assumption that $\boldsymbol{g}$ is twice differentiable and its Jacobian is full-rank everywhere

BERKELEY LAB

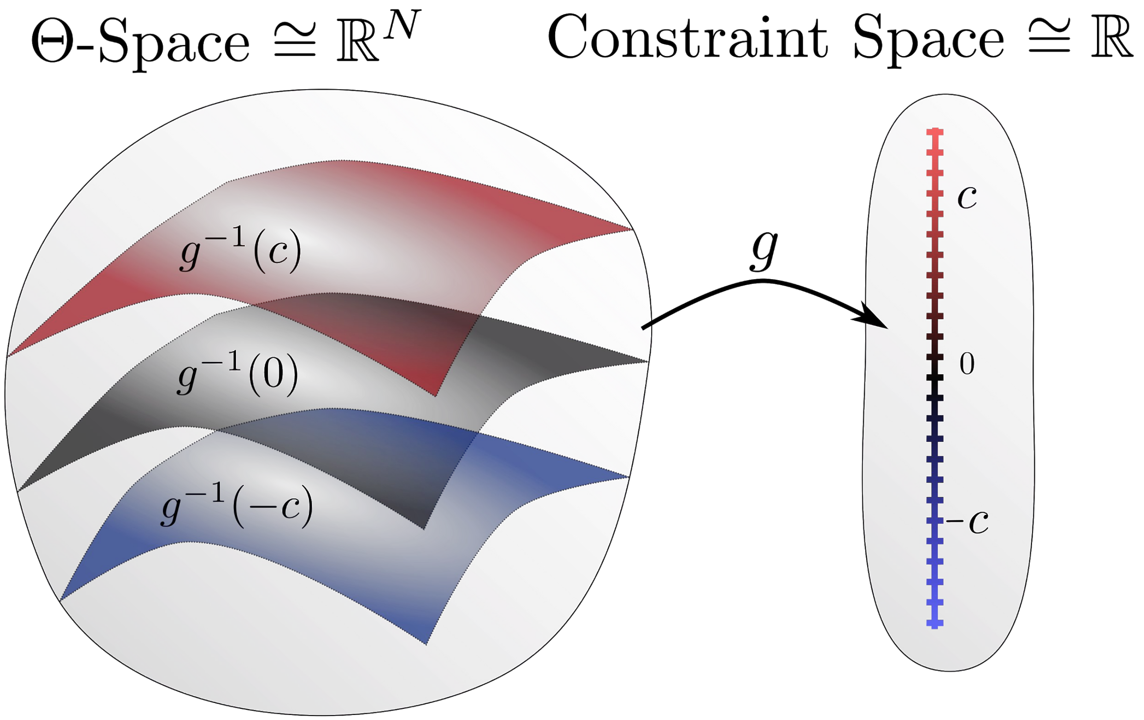
Optimization Problem

- Geometrically, NN training is a process which moves us along the dashed line
- Orange color fill here indicates how *negative* the minimization function is



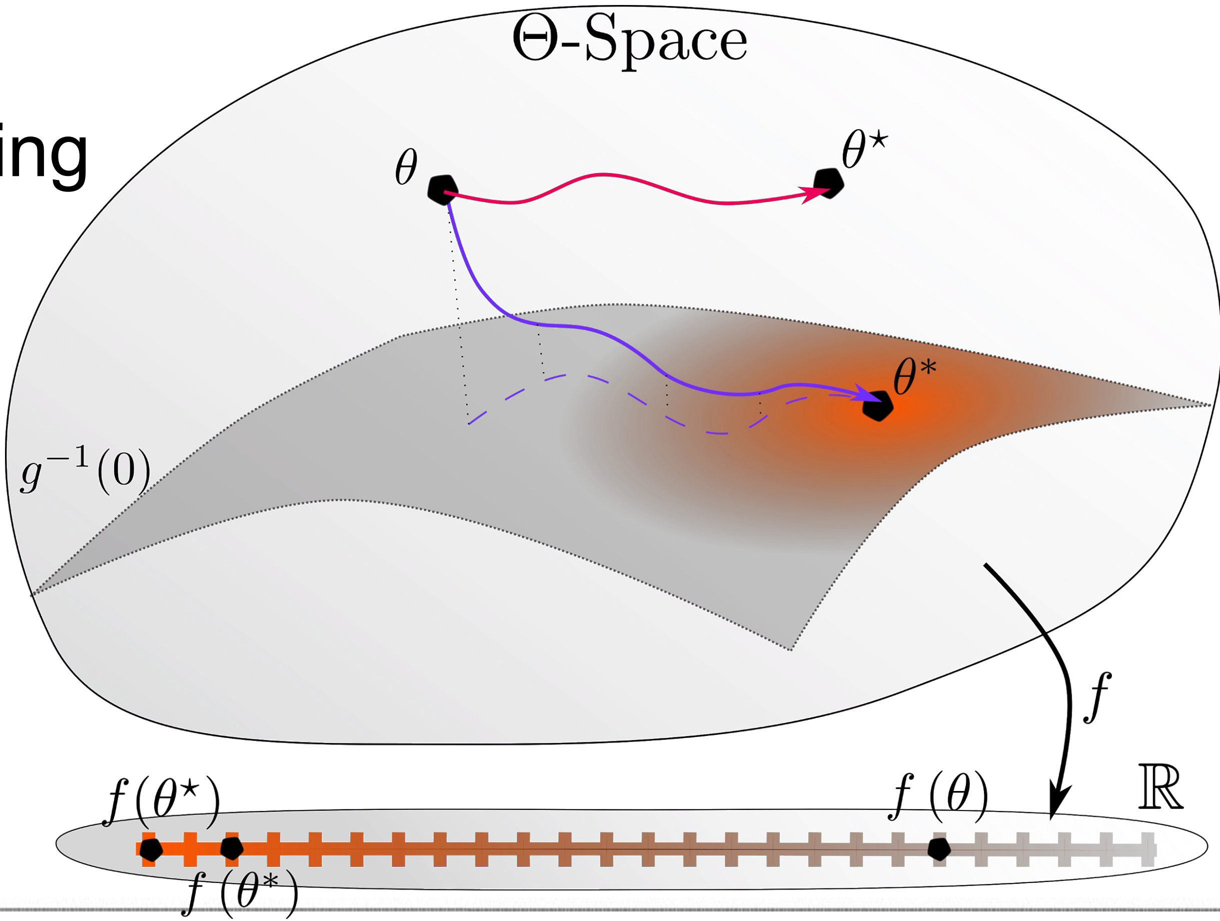
Constraint Manifold

- A second view of the optimization problem, this time from the viewpoint of the constraints
- For clarity in illustration, we assume the constraint function is real-valued
- Lightness roughly corresponds to optimality of minimization function



Constrained Training

- Normal training moves along red curve
- Constrained training moves along purple curve



BERKELEY LAB

Tanabe's Method

- Define the dynamical system:

$$\dot{\theta} = \frac{\mathrm{d}\theta}{\mathrm{d}t} = \Psi(\theta) = -\left(\mathrm{I} - J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) J(f(\theta))^{T} - J(\boldsymbol{g}(\theta))\boldsymbol{g}(\theta)$$

- $J(f(\theta))$ is the Jacobian of the function f at the point θ
- A^{+} is the Moore-Penrose Pseudoinverse of the matrix A
- If $\boldsymbol{g}$ is twice differentiable and $\boldsymbol{g}$'s first derivatives are linearly independent at θ (*i.e.* full-rank Jacobian), then $\boldsymbol{g}(\theta)$ will decay exponentially quickly

BERKELEY LAB

Proof of Tanabe's Method

$$\boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t}$$

$$\Longleftarrow$$

$$\dot{\boldsymbol{g}}(\theta(t)) = -\boldsymbol{g}(\theta(t))$$

$$\Longleftrightarrow$$

$$J(\boldsymbol{g}(\theta(t)))\frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = J(\boldsymbol{g}(\theta(t)))\dot{\theta} = -\boldsymbol{g}(\theta(t))$$

$$\left(J(\boldsymbol{g}(\theta))^{+}J(\boldsymbol{g}(\theta))\right)\dot{\theta} = -J(\boldsymbol{g}(\theta))^{+}\boldsymbol{g}(\theta(t))$$! Assumption

! The Jacobian (and therefore the pseudoinverse) needs to be full rank

Proof of Tanabe's Method

- True for any vector (under our assumptions):

$$v = \left(J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) v + \left(I - J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) v$$

- In particular,

$$\begin{aligned}\dot{\theta} &= \left(J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) \dot{\theta} + \left(I - J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) \dot{\theta} \\ &= -J(\boldsymbol{g}(\theta))^{+} \boldsymbol{g}(\theta(t)) + \left(I - J(\boldsymbol{g}(\theta))^{+} J(\boldsymbol{g}(\theta))\right) \dot{\theta}\end{aligned}$$

Proof of Tanabe's Method

$$\dot{\theta} = -J(\boldsymbol{g}(\theta))^{+}\boldsymbol{g}(\theta(t)) + \left(I - J(\boldsymbol{g}(\theta))^{+}J(\boldsymbol{g}(\theta))\right) h(\theta)$$

$$\Longrightarrow$$

$$\boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t}$$

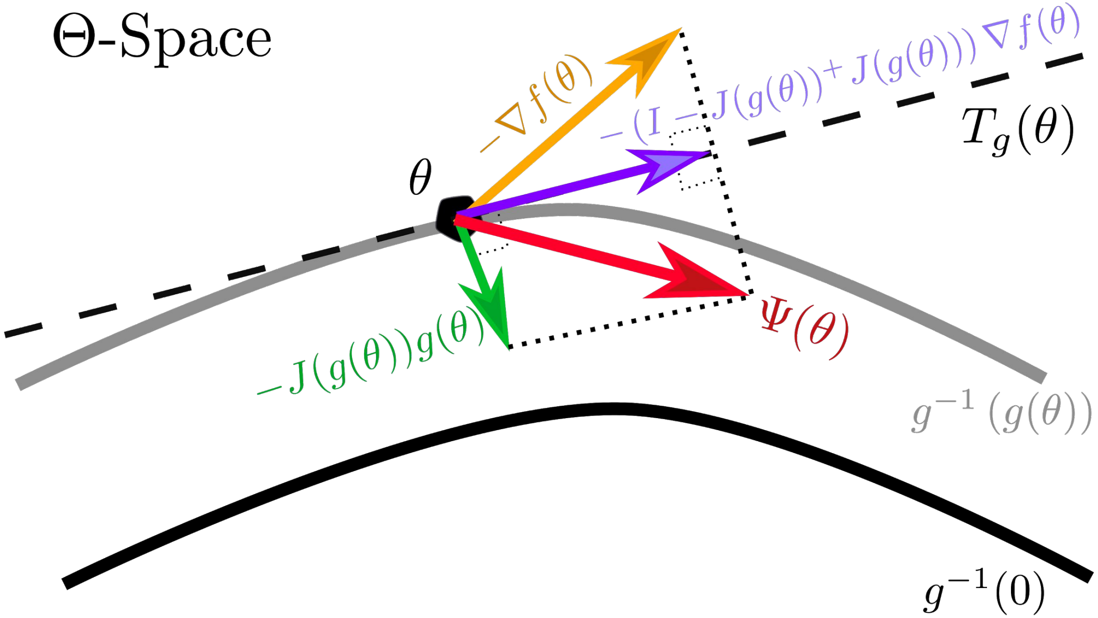
$$\dot{\theta} = \Psi(\theta) = -\left(I - J(\boldsymbol{g}(\theta))^{+}J(\boldsymbol{g}(\theta))\right) J(f(\theta))^{T} - J(\boldsymbol{g}(\theta))^{+}\boldsymbol{g}(\theta(t))$$

$$\Longrightarrow$$

$$\boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t} \qquad \square$$

Geometry of Tanabe's Method

- Yellow is the normal backpropagation vector
- Purple is the projection down to the tangent plane of the constraint surface
- Green is a correction term
- Red is the Tanabe's result



Implementing Tanabe's Method in Practice

- Tanabe's Method can also be written as

$$\dot{\theta}(t) = \frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = \Psi(\theta(t)) = -J(f(\theta(t)))^T - J(\boldsymbol{g}(\theta(t)))^T \Lambda(\theta(t)),$$

where $\Lambda(\theta(t)) = \left(J(\boldsymbol{g}(\theta(t)))J(\boldsymbol{g}(\theta(t)))^T\right)^{-1} \left(-J(\boldsymbol{g}(\theta(t)))J(f(\theta(t)))^T + \boldsymbol{g}(\theta(t))\right)$

- Equivalently, using a slightly different notation with $\lambda_i(\theta(t))$ for the components of $\Lambda(\theta(t))$

$$\Psi(\theta(t)) = -\nabla f(\theta(t)) - \sum_{i=1}^{M} \lambda_i(\theta(t)) * \nabla g_i(\theta(t))$$

Discretizing Tanabe's Method

- While Tanabe's Method is continuous, we need a discrete version. Applying a simple forward Euler method to

$$\Psi(\theta(t)) = -\nabla f(\theta(t)) - \sum_{i=1}^{M} \lambda_i(\theta(t)) * \nabla g_i(\theta(t))$$

gives

$$\theta_k = \theta_{k-1} + \eta * \Psi(\theta_{k-1}) = \theta_{k-1} - \eta * \left(\nabla f(\theta_{k-1}) + \sum_{i=1}^{M} \lambda_i(\theta_{k-1}) \nabla g_i(\theta_{k-1}) \right)$$

Tanabe's Method and Backpropagation

- Normal backpropagation on the loss function $\mathcal{L}$ gives

$$\theta_k = \theta_{k-1} - \eta * \nabla\mathcal{L}(\theta_{k-1})$$

- Comparing this to our forward Euler discretization

$$\theta_k = \theta_{k-1} + \eta * \Psi(\theta_{k-1}) = \theta_{k-1} - \eta * \left(\nabla f(\theta_{k-1}) + \sum_{i=1}^{M} \lambda_i(\theta_{k-1})\nabla g_i(\theta_{k-1})\right)$$

tells us we need

$$\mathcal{L}(\theta_k) = f(\theta_k) + \sum_{i=1}^{M} \mathtt{nograd}(\lambda_i(\theta_k)) * g_i(\theta_k)$$

Implementing Tanabe's Method in Practice

- Result: We can simply define a new loss function to do constrained optimization!

$$\mathcal{L}(\theta_k) = f(\theta_k) + \sum_{i=1}^{M} \texttt{nograd}(\lambda_i(\theta_k)) * g_i(\theta_k)$$

$$\text{where} \quad \lambda_i(\theta_k) = \Lambda(\theta_k)_i$$

$$\Lambda(\theta_k) = \left(J(\boldsymbol{g}(\theta_k)) \cdot J(\boldsymbol{g}(\theta_k))^T\right)^{-1} \left(-J(\boldsymbol{g}(\theta_k)) \cdot J(f(\theta_k))^T + \boldsymbol{g}(\theta_k)\right)$$

One Final Detail: What About Minibatch Backprop?

- Two options:

1. Take average over entire update:

$$\mathcal{L}(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k) + \sum_{i=1}^{M} \texttt{nograd}(\lambda_i(x_B;\theta_k)) * g_i(x_B;\theta_k)\right]$$

One Final Detail: What About Minibatch Backprop?

- Two options:

2. First average the loss function and apply some reduction to the constraints

$$\mathcal{L}(x_B;\theta_k) = \bar{f}(x_B;\theta_k) + \sum_{j=1}^{M} \texttt{nograd}\left(\tilde{\lambda}_j(x_B;\theta_k)\right) * \tilde{g}_j(x_B;\theta_k)$$

$$\bar{f}(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k)\right] = \frac{1}{B}\sum_{i=1}^{B} f(x_i;\theta_k)$$

$$\tilde{\boldsymbol{g}}(x_B;\theta_k) = \texttt{Reduce}_{x_i\in x_B}(\boldsymbol{g}(x_B;\theta_k)) = \quad e.g. \quad \sum_{i=1}^{B}\sum_{j=1}^{M}(g_j(x_i;\theta_k))^2$$

BERKELEY LAB

One Final Detail: What About Minibatch Backprop?

1. Fully Constrained Method
   a. Pro: Seems more “correct”
   b. Con: Will scale poorly, since many sets of multipliers must be computed
2. Reduction Method
   a. Pro: Scales much better, since it doesn’t depend on batch size
   b. Con: Requires some ad-hoc reduction / error function to be applied to the constraints

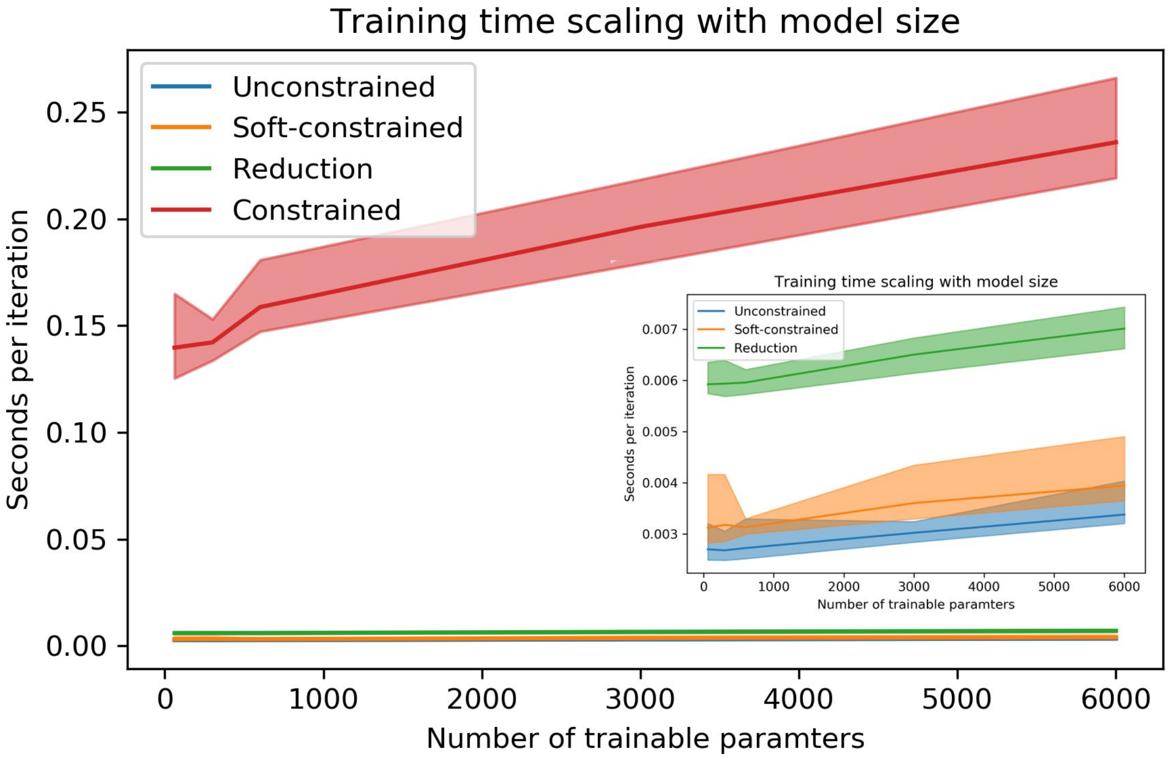
- Con for both methods: requires computing the Jacobians of the loss and constraint functions over ALL parameters in the network
  - Only feasible for very small networks

$$\Lambda(\theta_k) = \left(J(\boldsymbol{g}(\theta_k)) \cdot J(\boldsymbol{g}(\theta_k))^T\right)^{-1} \left(-J(\boldsymbol{g}(\theta_k)) \cdot J(f(\theta_k))^T + \boldsymbol{g}(\theta_k)\right)$$

BERKELEY LAB

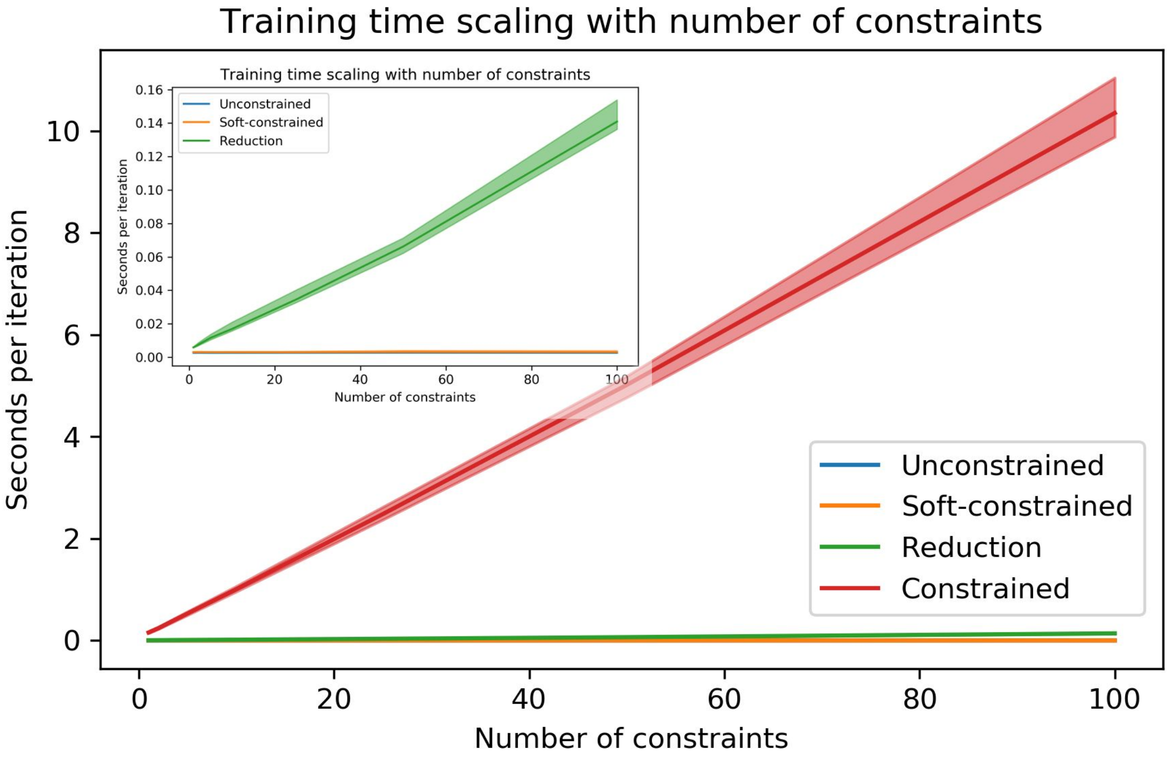
Time Complexity

- Batch size: 100
- Real-valued constraint



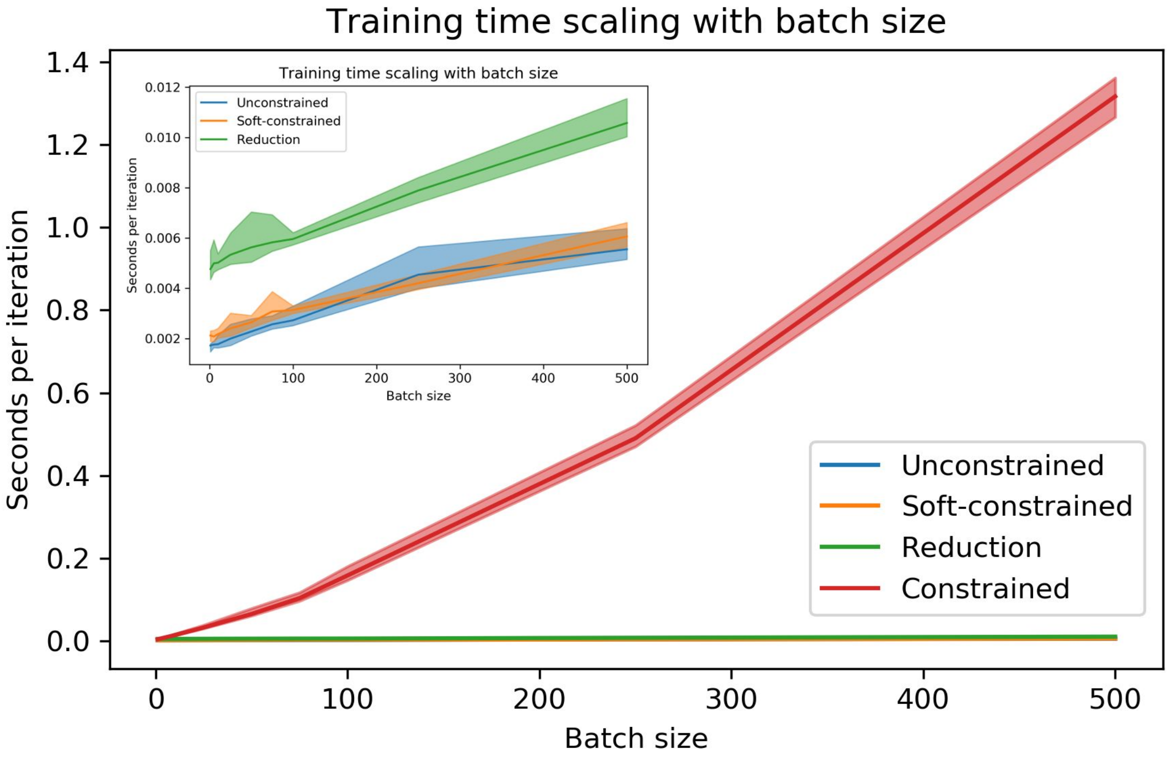
Time Complexity

- Batch size: 100
- Model size: ~600 trainable parameters



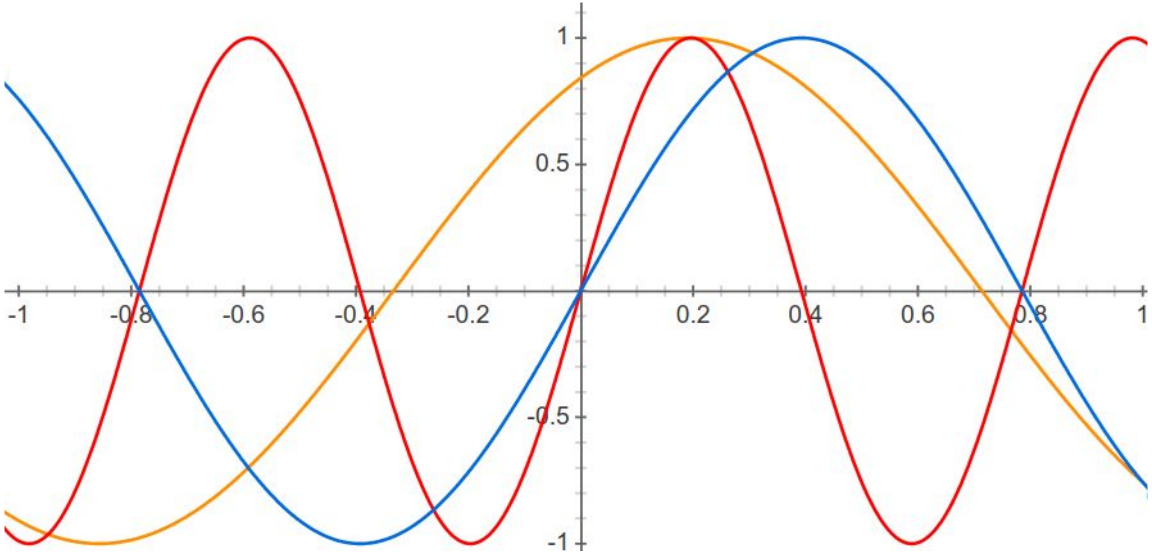
Time Complexity

- Real-valued constraint
- Model size: ~600 trainable parameters



Experimental Design

$$u = A \sin(kx + \phi)$$



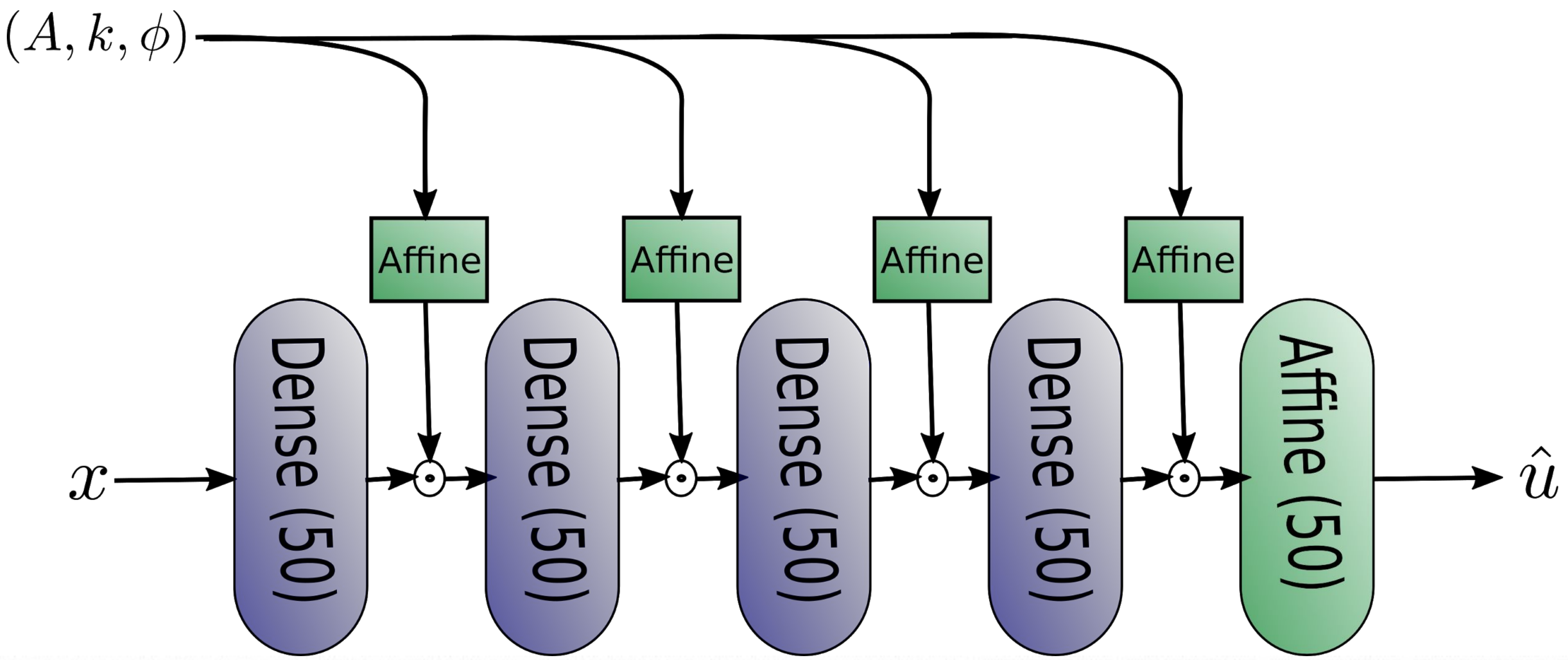
- Training data:
  - 1000 sine waves with different parameters:

    A: $[0.2, 5.0]$

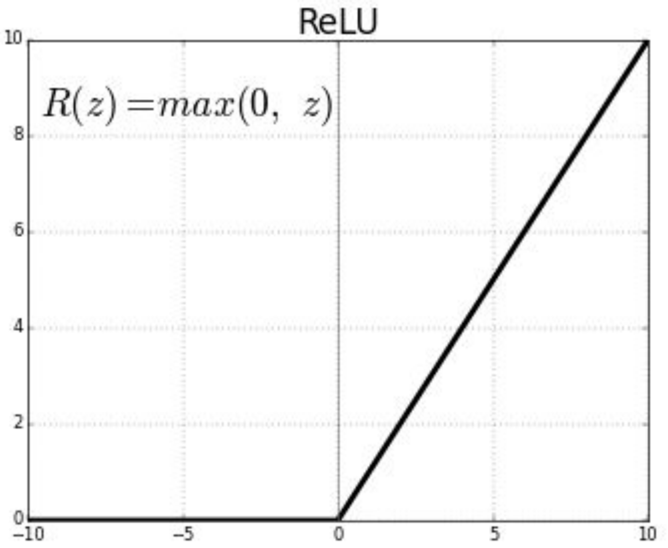
    k: $[0.4 * \pi, 10 * \pi]$

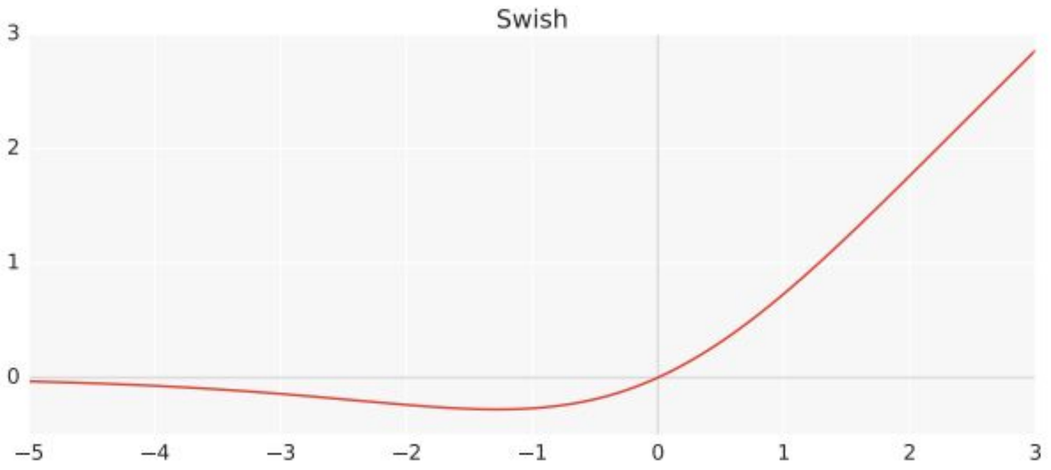
    ϕ: $[-0.5, 0.5]$
  - 50 random x-values per curve
- Batch size: 1000
- Learning Rate: 10^{-3}
- Constraint: Helmholtz Equation: $\left(\nabla^2 + k^2\right) u = 0$

Neural Network Architecture



Activation Function

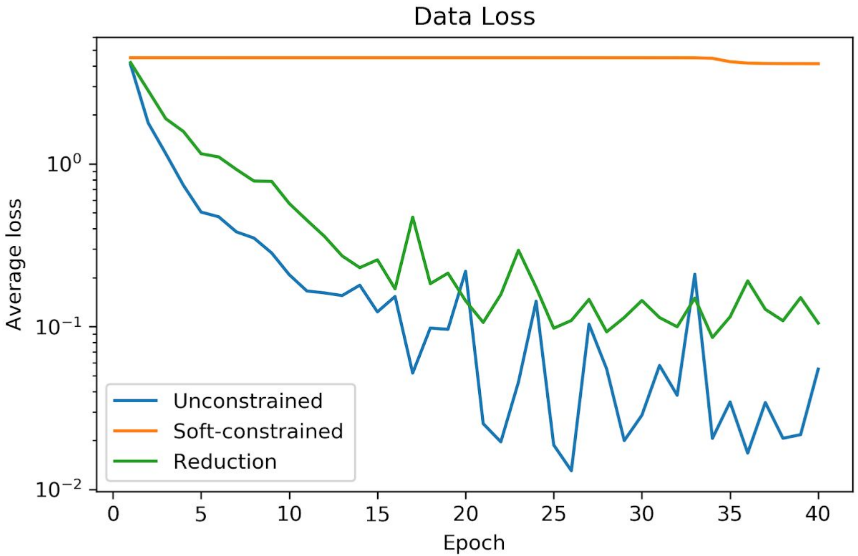


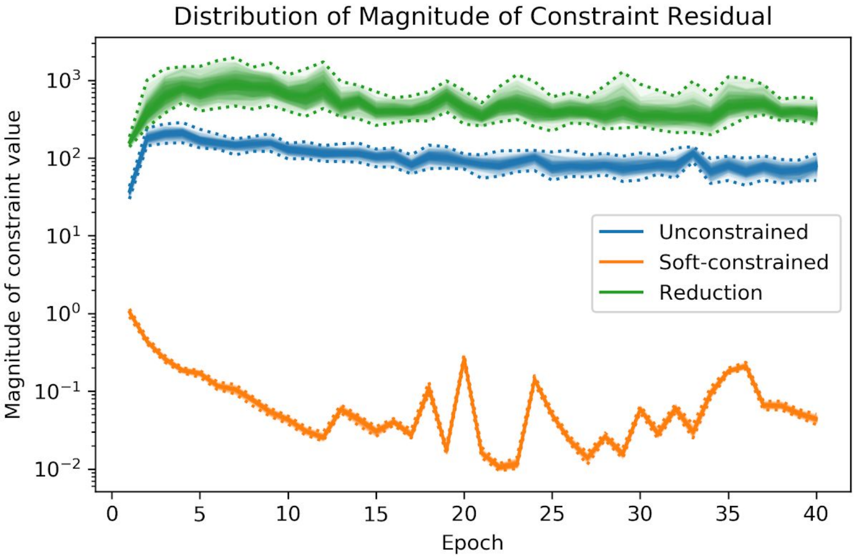


https://medium.com/@kanchansarkar/relu-not-a-differentiable-function-why-used-in-gradient-based-optimization-7fef3a4cecec

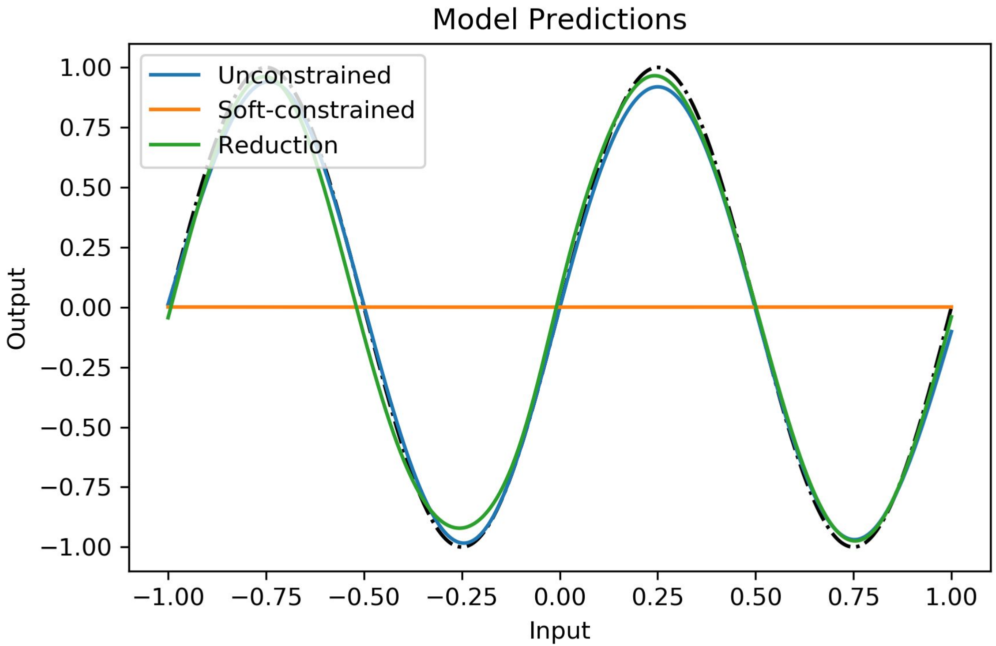
https://medium.com/@neuralnets/swish-activation-function-by-google-53e1ea86f820

Training Characteristics





Model Predictions



Nonlinear Projection

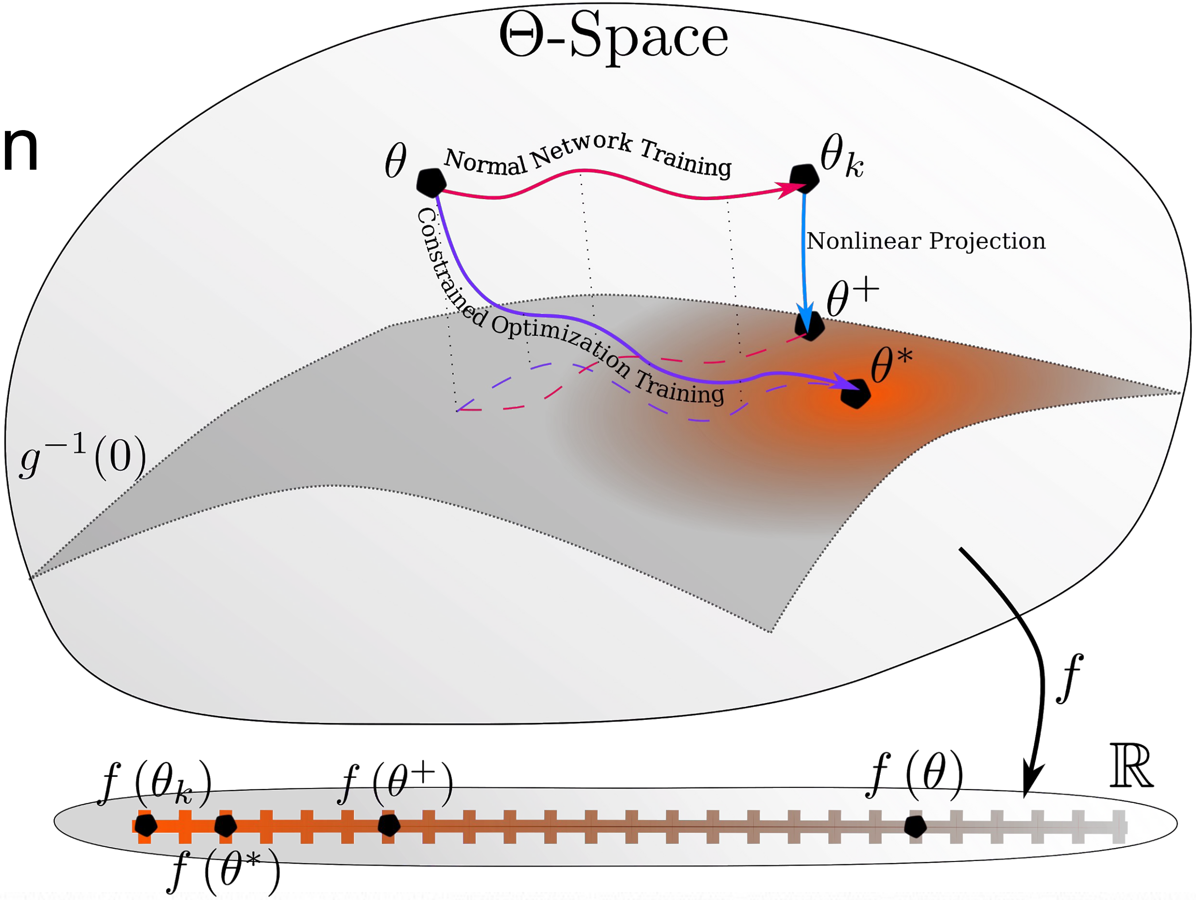
Nonlinear Projection

- Idea: take a trained neural network and project its parameters to the constraint manifold after training is complete:

$$\theta^{+} = \underset{\substack{\theta \in \Theta \\ \boldsymbol{g}(\theta) = \mathbf{0}}}{\operatorname{argmin}} \|\theta_k - \theta\|$$

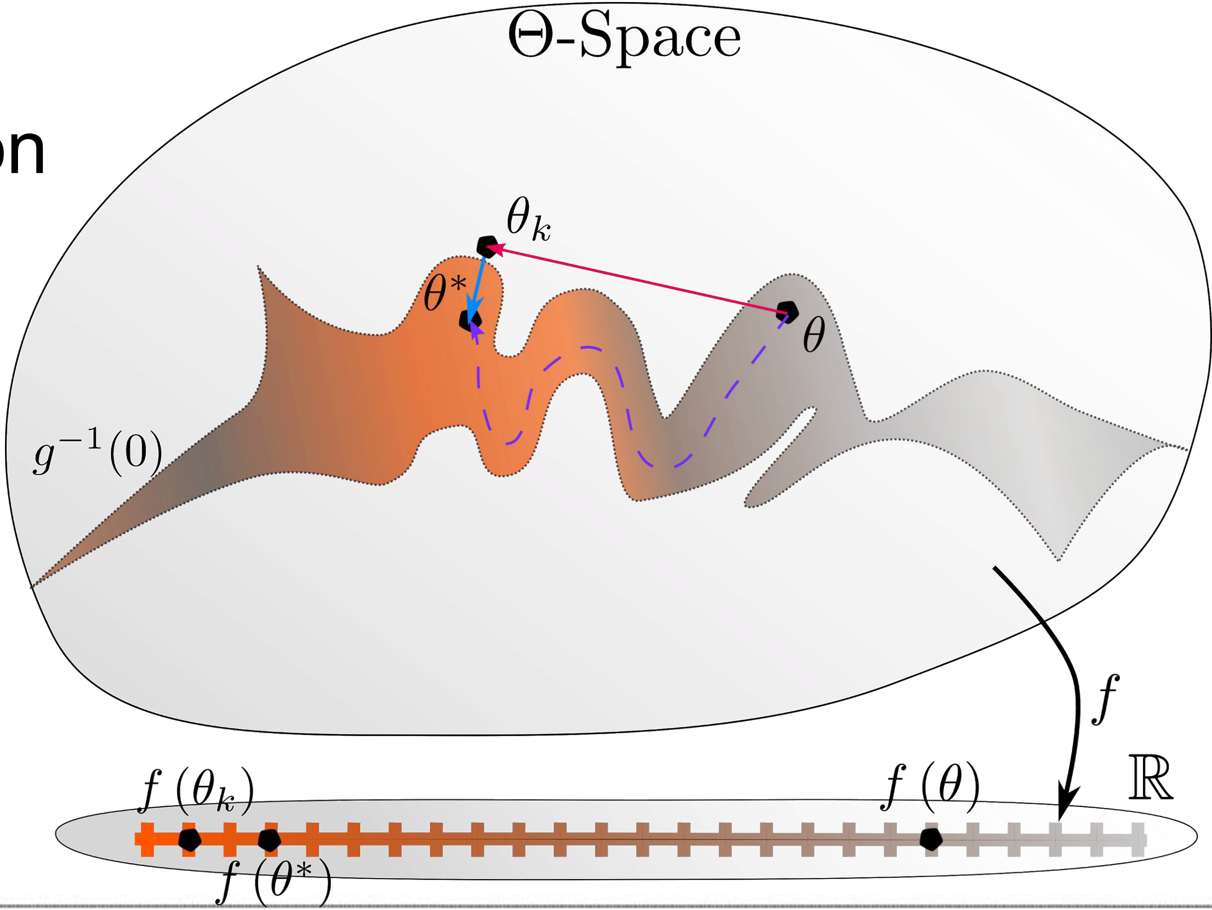
Method Comparison

- Normal training + projection is much faster than constrained optimization
- Can yield different final parameters
- Can be applied to a model which is already trained!



Method Comparison

- If the constraint manifold is “warped”, normal training + projection has shorter distance than constrained optimization



Experimental Design

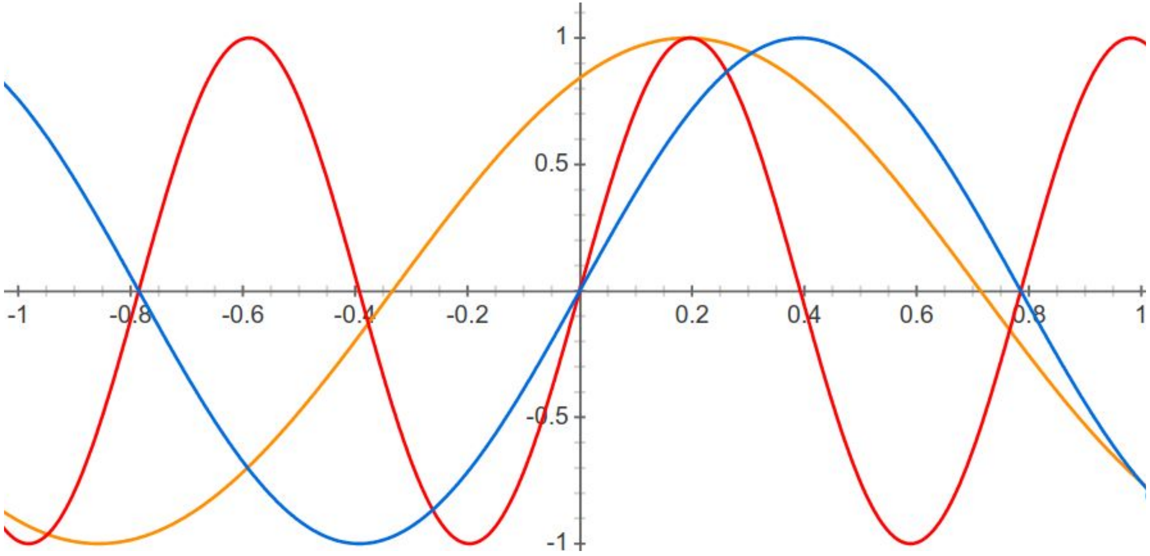
$$u = A\sin(kx + \phi)$$

- Training data:
  - 1000 sine waves with different parameters:

    A: $[0.2, 5.0]$

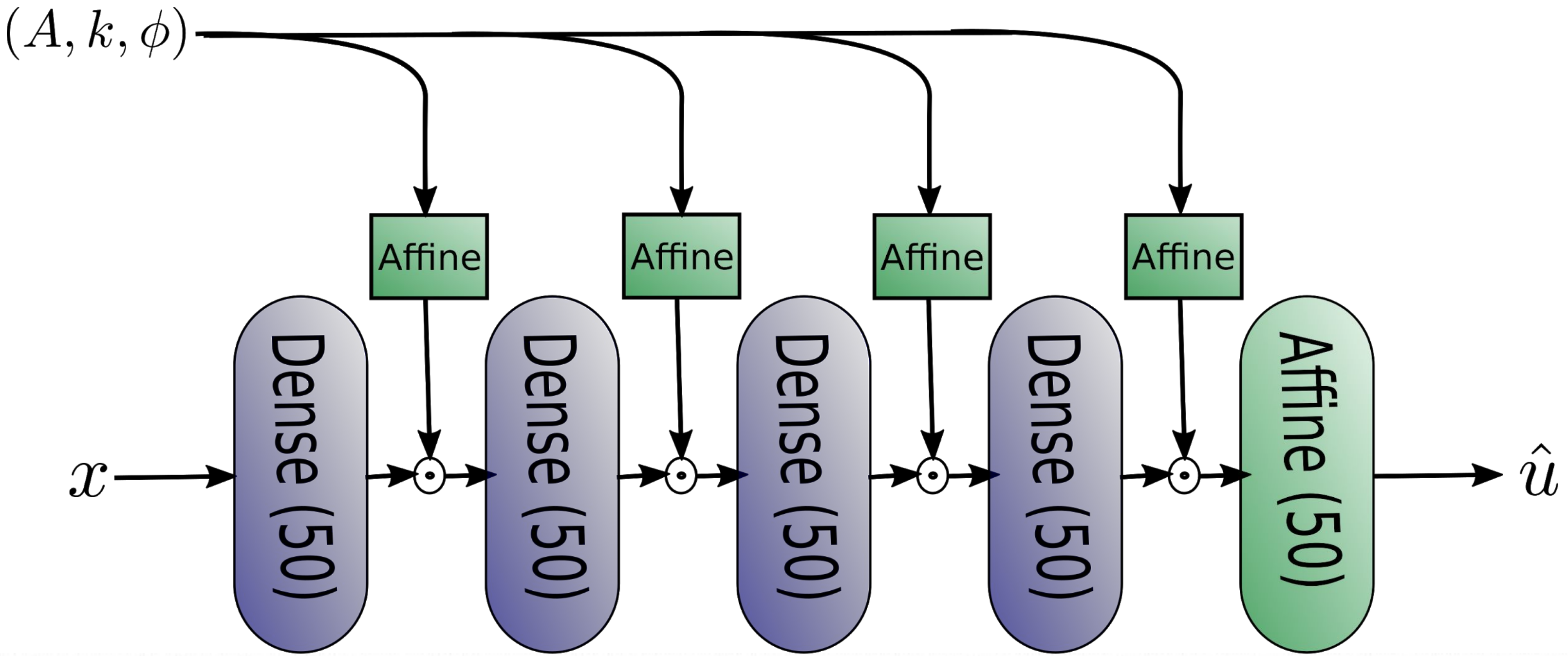
    k: $[0.4 * \pi, 10 * \pi]$

    ϕ: $[-0.5, 0.5]$
  - 50 random x-values per curve
- Batch size: 1000
- Learning Rate: 10^{-3}
- Projection LR: 10^{-4}
- Constraint: Helmholtz Equation: $\left(\nabla^2 + k^2\right) u = 0$

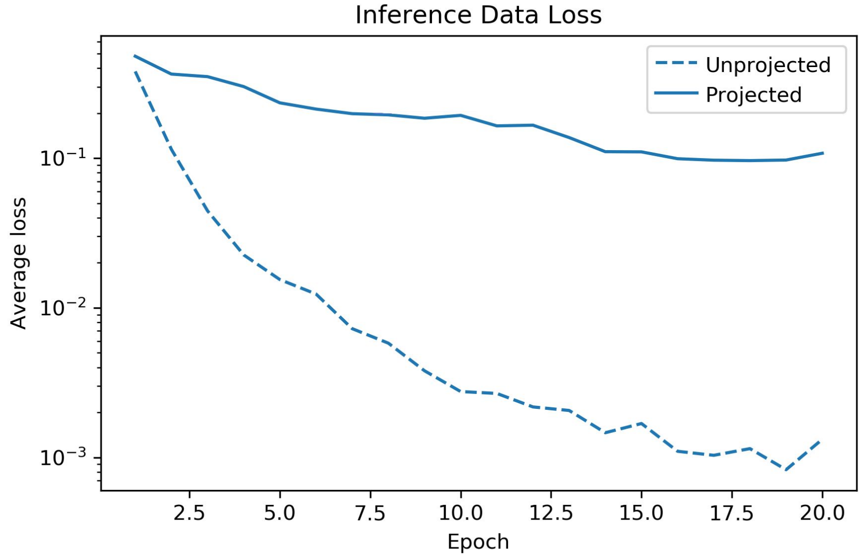


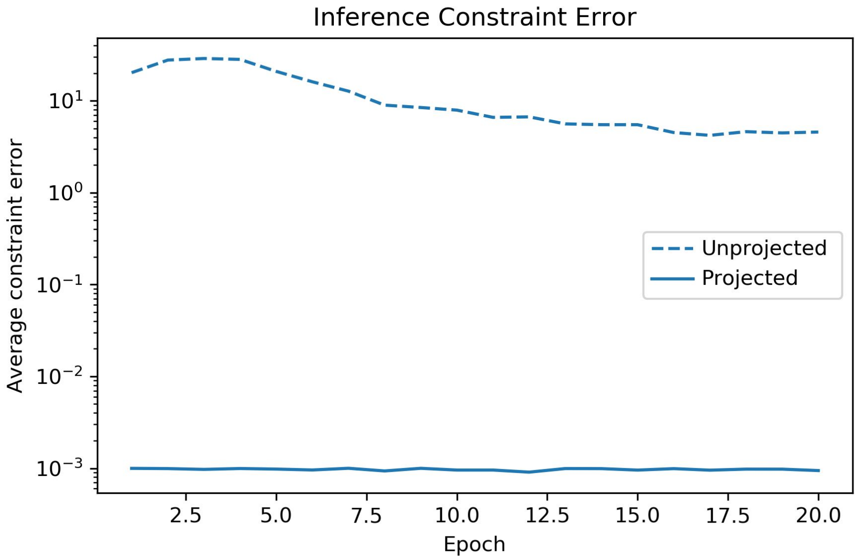
BERKELEY LAB

Neural Network Architecture

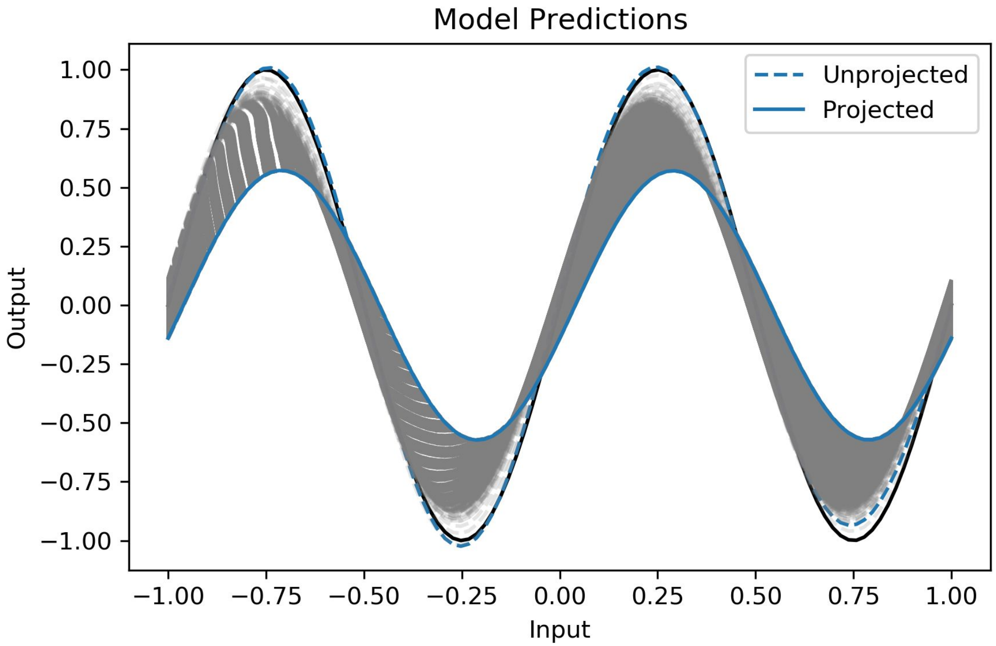


Nonlinear Projection Characteristics

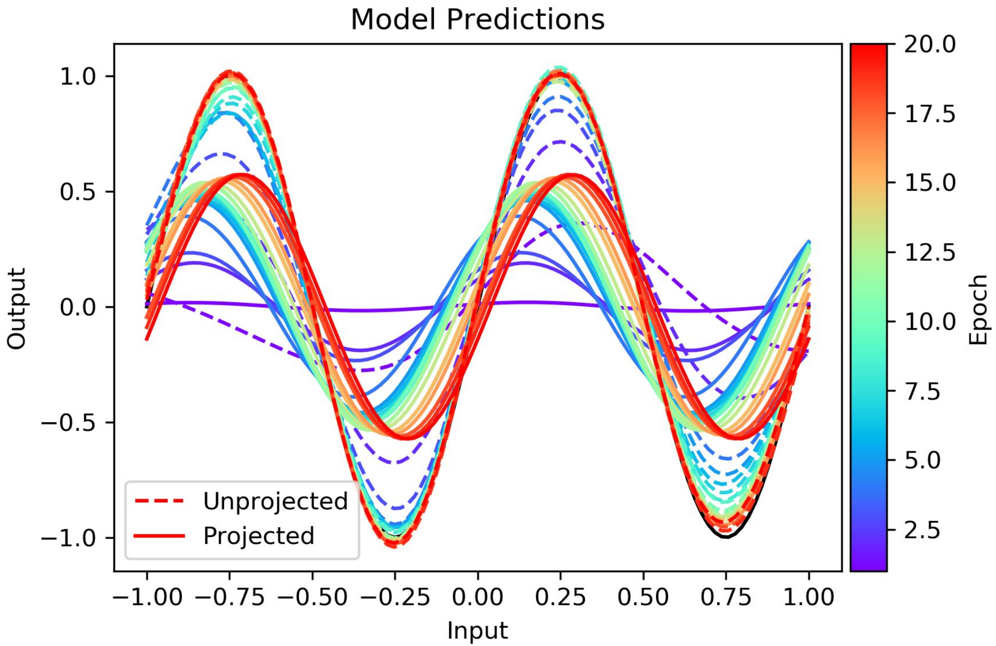




Projection Process



Model Predictions



Conclusion

- Constrained optimization is too slow to be used in practice (except maybe for really small neural networks)
  - Provides a good framework for designing and developing loss functions
  - Does come with good theoretical guarantees
- Nonlinear projection isn’t guaranteed to reach a minimum of the constrained optimization problem
  - Fast enough to be used
  - Interesting as an interpretability method
- Possible solution: combine the two
  - Use constrained optimization only during projection

Conclusion and Future Work

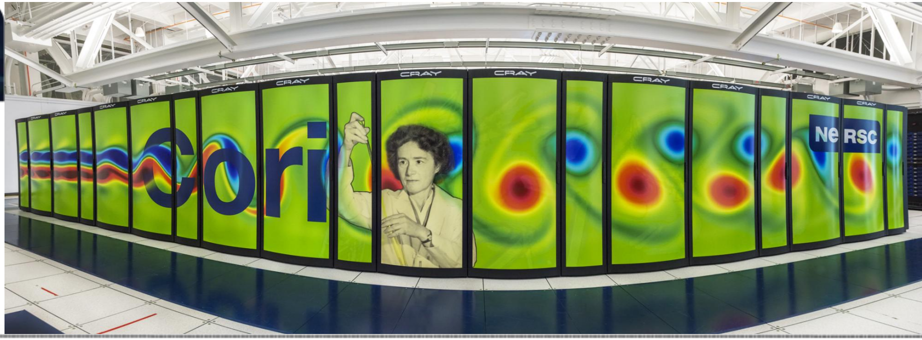
- Another possible solution: Rephrase the problem to use Spectral Methods
  - Work in Fourier domain (i.e. frequency space)
  - Comes with guarantees that all L_2 functions (square integrable) can be represented
  - PDEs are often easier to write down
    - Easier for neural network to disentangle representations
- Current project at Berkeley Labs: Use NNs in combination with PDE Solvers
  - Neural network works as a data-driven approximator
  - PDE Solver corrects the solution to be exact
  - Also equivalent to preconditioning PDE Solvers
    - Makes PDE Solvers faster

Acknowledgements

- Mr. Prabhat
- Karthik Kashinath
- Chiyu “Max” Jiang

- NERSC:

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.



References

Bar-Sinai, Y., Hoyer, S., Hickey, J., and Brenner, M. P. (2018). Data-driven discretization: a method for systematic coarse graining of partial differential equations. *arXiv preprint arXiv:1808.04930*.

Branin, F. H. (1972). Widely convergent method for finding multiple solutions of simultaneous nonlinear equations. *IBM Journal of Research and Development*, 16(5):504–522.

Broyden, C. (1969). A new method of solving nonlinear simultaneous equations. *The Computer Journal*, 12(1):94–99.

Broyden, C. G. (1965). A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593.

Canuto, C., Hussaini, M. Y., Quarteroni, A., and Zang, T. A. (2006). *Spectral methods*. Springer.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 6571–6583. Curran Associates, Inc.

Ha, D., Dai, A., and Le, Q. V. (2016). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Karras, T., Laine, S., and Aila, T. (2018). A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*.

Long, Z., Lu, Y., and Dong, B. (2018). Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network. *arXiv preprint arXiv:1812.04426*.

References

Long, Z., Lu, Y., Ma, X., and Dong, B. (2017). Pde-net: Learning pdes from data. *arXiv preprint arXiv:1710.09668*.

Márquez-Neila, P., Salzmann, M., and Fua, P. (2017). Imposing hard constraints on deep networks: Promises and limitations. *arXiv preprint arXiv:1706.02025*.

Platt, J. C. and Barr, A. H. (1988). Constrained differential optimization. In *Neural Information Processing Systems*, pages 612–621.

Raissi, M. (2018). Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research*, 19(1):932–955.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 7.

Straeter, T. A. (1971). On the extension of the davidon-broyden class of rank one, quasi-newton minimization methods to an infinite dimensional hilbert space with applications to optimal control problems.

Tanabe, K. (1980). A geometric method in nonlinear programming. *Journal of Optimization Theory and Applications*, 30(2):181–210.

Questions?

https://github.com/gelijergensen/Constrained-Neural-Nets-Workbook