n-TangentProp: A Quasilinear Algorithm for Taking Derivatives of Neural Networks

Kyle R. Chickering
UC Davis

1. Motivation
2. PINNs + Neural Networks
3. Autodiff
4. TangentProp + n-TangentProp
5. Results
6. Conclusions

1. Motivation
2. PINNs + Neural Networks
3. Autodiff
4. TangentProp + n-TangentProp
5. Results
6. Conclusions

Paper online, work under Dr. P. Simard in Feb. of last year

Motivation I

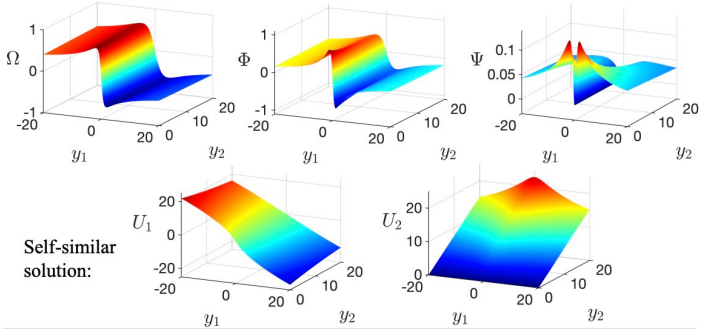
- The Navier-Stokes Existence & Smoothness (T. Tao in Notices)

Motivation I

- The Navier-Stokes Existence & Smoothness (T. Tao in Notices)
- Self-Similar Solutions tell us about finite-time Breakdown

Motivation I

- The Navier-Stokes Existence & Smoothness (T. Tao in Notices)
- Self-Similar Solutions tell us about finite-time Breakdown



Boussinesq

$$
\begin{aligned}
\Omega + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Omega &= \Phi\,, \\
(2 + \partial_{y_1} U_1)\Phi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Phi &= -\partial_{y_1} U_2 \Psi\,, \\
(2 + \partial_{y_2} U_2)\Psi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Psi &= -\partial_{y_2} U_1 \Phi\,, \\
\operatorname{div} \mathbf{U} &= 0\,.
\end{aligned}
\tag{3}
$$

Hard to solve numerically!

From Wang, Lai, Gómez-Serrano, Buckmaster '23

Motivation II

- The "boundary conditions" for self-similar solutions look like smoothness constraints, i.e. $U \in C^5(B_\varepsilon(x_0))$ for a point x_0

Motivation II

- The "boundary conditions" for self-similar solutions look like smoothness constraints, i.e. $U \in C^5(\mathbb{B}_\varepsilon(x_0))$ for a point x_0
- Exact equation is unknown! Must find $\lambda, U, \Omega, \Psi, \Phi$ all at once!

$$
\begin{aligned}
\Omega + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Omega &= \Phi\,, \\
(2 + \partial_{y_1} U_1)\Phi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Phi &= -\partial_{y_1} U_2 \Psi\,, \\
(2 + \partial_{y_2} U_2)\Psi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Psi &= -\partial_{y_2} U_1 \Phi\,, \\
\operatorname{div} \mathbf{U} &= 0\,.
\end{aligned} \tag{3}
$$

We don't know what λ is!

Motivation II

- The "boundary conditions" for self-similar solutions look like smoothness constraints, i.e. $U \in C^5(\mathbb{B}_\varepsilon(x_0))$ for a point x_0
- Exact equation is unknown! Must find $\lambda, U, \Omega, \Psi, \Phi$ all at once!

$$
\begin{aligned}
\Omega + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Omega &= \Phi\,, \\
(2 + \partial_{y_1} U_1)\Phi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Phi &= -\partial_{y_1} U_2 \Psi\,, \\
(2 + \partial_{y_2} U_2)\Psi + ((1+\lambda)\mathbf{y} + \mathbf{U}) \cdot \nabla \Psi &= -\partial_{y_2} U_1 \Phi\,, \\
\operatorname{div} \mathbf{U} &= 0\,.
\end{aligned} \tag{3}
$$

We don't know what λ is!

- PINNs!

PINN's Neural Networks

Neural Networks I

- Repeated functional composition

$$u = \sigma(W^n \cdot \sigma(W^{n-1} \cdot \sigma(\cdots \sigma(W^1 \cdot x))))$$

- Nonlinear activation σ

Neural Networks I

- Repeated functional composition

$$u = \sigma(W^n \cdot \sigma(W^{n-1} \cdot \sigma(\cdots \sigma(W^1 \cdot x))))$$

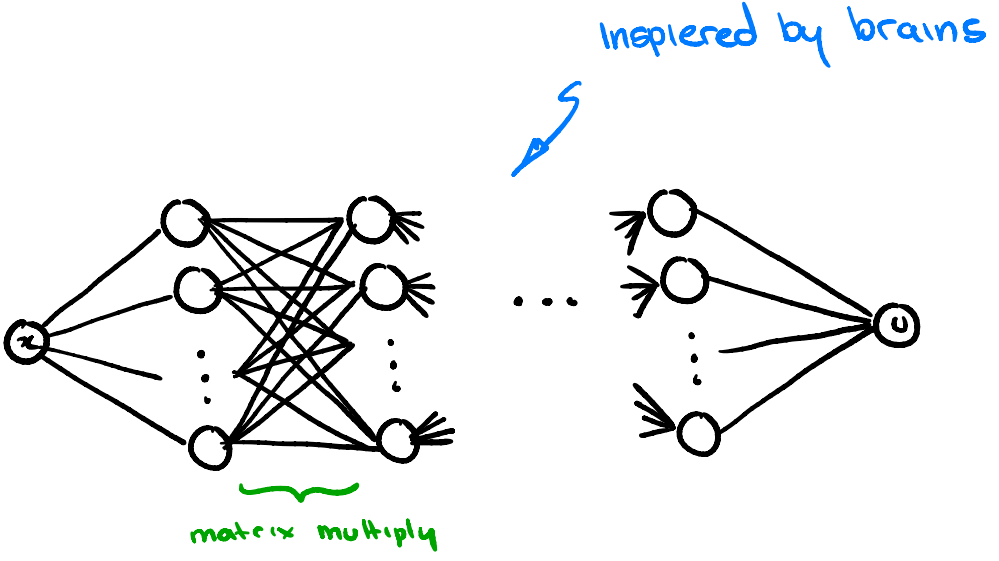
- Nonlinear activation σ
- Nonlinear is important!

Neural Networks I

- Repeated functional composition

$$u = \sigma(W^n \cdot \sigma(W^{n-1} \cdot \sigma(\cdots \sigma(W^1 \cdot x))))$$

- Nonlinear activation σ
- Nonlinear is important!



Neural Networks II

- Universal Functional approximator

Neural Networks II

- Universal functional approximator
- As width $\longrightarrow \infty$ we can approximate any continuous function on a compact domain

Neural Networks II

- Universal Functional approximator
- As width $\longrightarrow \infty$ we can approximate any continuous function on a compact domain
- Does not tell us <u>how</u> to get approximation

Neural Networks III

- Suppose I have a labled dataset $X = \{(x_i, y_i)\}_{i \in \mathcal{I}}$, & I want a function to approximate or generalize X, i.e. $f(x_i) \sim y_i \ \forall \ i \in \mathcal{I}$.

Neural Networks III

- Suppose I have a labled dataset $X = \{(x_i, y_i)\}_{i \in \mathcal{I}}$, & I want a function to approximate or generalize X, i.e. $f(x_i) \sim y_i \ \forall \ i \in \mathcal{I}$.
- Neural net: update weights W.

Neural Networks III

- Suppose I have a labled dataset $X = \{(x_i, y_i)\}_{i \in \mathcal{I}}$, & I want a function to approximate or generalize X, i.e. $f(x_i) \sim y_i \ \forall \ i \in \mathcal{I}$.
- Neural net: update weights W.
- Loss: $$L(v_W) = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} |v_W(x_i) - y_i|^2$$

Neural Networks III

- Suppose I have a labled dataset $X = \{(x_i, y_i)\}_{i \in \mathcal{I}}$, & I want a function to approximate or generalize X, i.e. $f(x_i) \sim y_i \ \forall \ i \in \mathcal{I}$.
- Neural net: update weights W.
- Loss: $$L(u_W) = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} |u_W(x_i) - y_i|^2$$
- Update params: $$W \leftarrow W - \alpha \nabla_W L(u_W)$$
- Gradient descent: $\nabla_W L(u_W) \to 0$ means critical pt.

PINNs

- Neural Networks are C^∞ functions

PINNs

- Neural Networks are C^∞ functions
- If u_θ is a neural network, we can compute

  $\frac{d^n}{dx^n} u_\theta$ exactly.

PINNs

- Neural Networks are C^∞ functions
- If u_θ is a neural network, we can compute

  $\frac{d^n}{dx^n} u_\theta$ exactly.

- If $F(\partial^\alpha u; x)$ a differential equation, run gradient decent

$$L(u_\theta) = \frac{1}{N} \sum_{k=1}^{N} |F(\partial^\alpha u_\theta; x_k)|^2 + BC$$

PINNs

- Neural Networks are C^∞ functions
- If u_θ is a neural network, we can compute

  $\frac{d^n}{dx^n} u_\theta$ exactly.

- If $F(\partial^\alpha u; x)$ a differential equation, run gradient decent

$$L(u_\theta) = \frac{1}{N} \sum_{k=1}^{N} |F(\partial^\alpha u_\theta; x_k)|^2 + BC$$

- NNs as functional approximator, $u_\theta \longrightarrow u$ during training!

Accuracy Gains

- Sobolev Loss

$$L_{Sob.}^{(m)}(v_\theta) = \frac{1}{N} \sum_{k=1}^{N} \sum_{j=0}^{m} a_j \left| \nabla_x^j F(a^\alpha v_\theta; x_k) \right|^2 + BC$$

Accuracy Gains

- Sobolev Loss

$$L_{Sob.}^{(m)}(u_\theta) = \frac{1}{N}\sum_{k=1}^{N}\sum_{j=0}^{m} a_j \left|\nabla_x^j F(a^\alpha u_\theta; x_k)\right|^2 + BC$$

Key takeaway: More enforced derivatives gives better accuracy & training efficiency

(Maddu et al. '22)

Accuracy Gains

- Sobolev Loss

$$L_{Sob.}^{(m)}(u_\theta) = \frac{1}{N}\sum_{k=1}^{N}\sum_{j=0}^{m} a_j \left|\nabla_x^j F(a^\alpha u_\theta; x_k)\right|^2 + BC$$

Key takeaway: More enforced derivatives gives better accuracy & training efficiency (Maddu et al. '22)

→ More Derivatives!

Do PINNs Just Suck?

- Hard to train (Hessian)

Do PINNs Just Suck?

- Hard to train (Hessian)
- Inefficient to train, expensive to train

Do PINNs Just Suck?

- Hard to train (Hessian)
- Inefficient to train, expensive to train
- No rigorous error analysis

Do PINNs Just Suck?

- Hard to train (Hessian)
- Inefficient to train, expensive to train
- No rigorous error analysis
- "Cold Water" Papers

PINN Use Cases

- Combined Forward-Inverse Problems
  - → measurements from a fluid experiment
  - → Solve N.S. & viscosity simultaneously

PINN Use Cases

- Combined Forward-Inverse Problems
  - → measurements from a fluid experiment
  - → Solve N.S. & viscosity simultaneously
- Auxillary Conditions
  - → For example: Enforcing smoothness

PINN Use Cases

- Combined Forward-Inverse Problems
  - → measurements from a fluid experiment
  - → Solve N.S. & viscosity simultaneously
- Auxillary Conditions
  - → For example: Enforcing smoothness
- If you can do it easily w/o a PINN, you probably don't need a PINN!

Autodiff

Autodiff

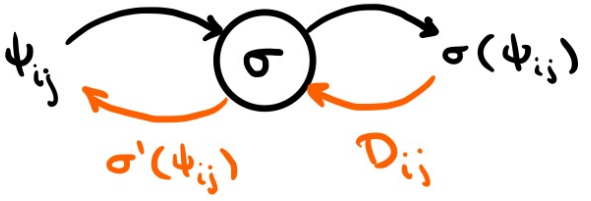
- How do we actually compute the derivatives?

Autodiff

- How do we actually compute the derivatives?
- Computational Graphs (DAG)

Autodiff

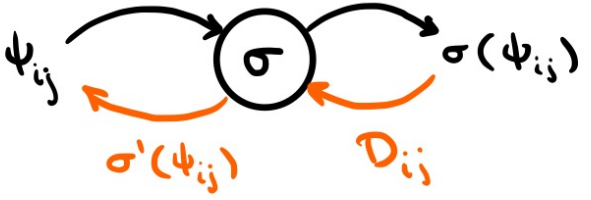
- How do we actually compute the derivatives?
- Computational Graphs (DAG)



- Forward: $\sigma(\psi)$

Autodiff

- How do we actually compute the derivatives?
- Computational Graphs (DAG)



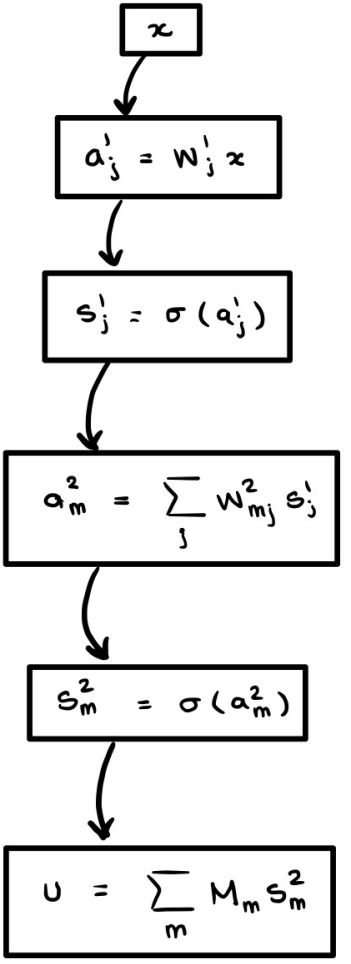
- Forward: $\sigma(\psi)$
- Backwards: $D \odot \sigma'(\psi)$

Computation Graph for NN

$$u = M \cdot \sigma(W^2 \cdot \sigma(W^1 x))$$

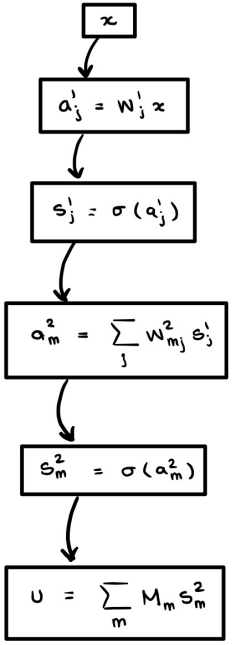
Computation Graph for NN

$$U = M \cdot \sigma(W^2 \cdot \sigma(W^1 x))$$



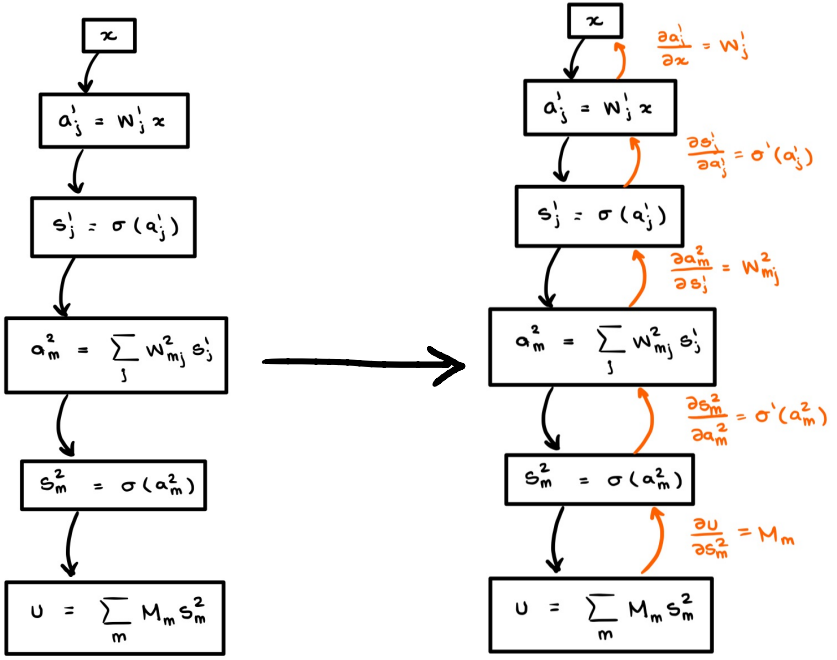
Autodiff Algorithm

- Forwards → Backwards → New Graph



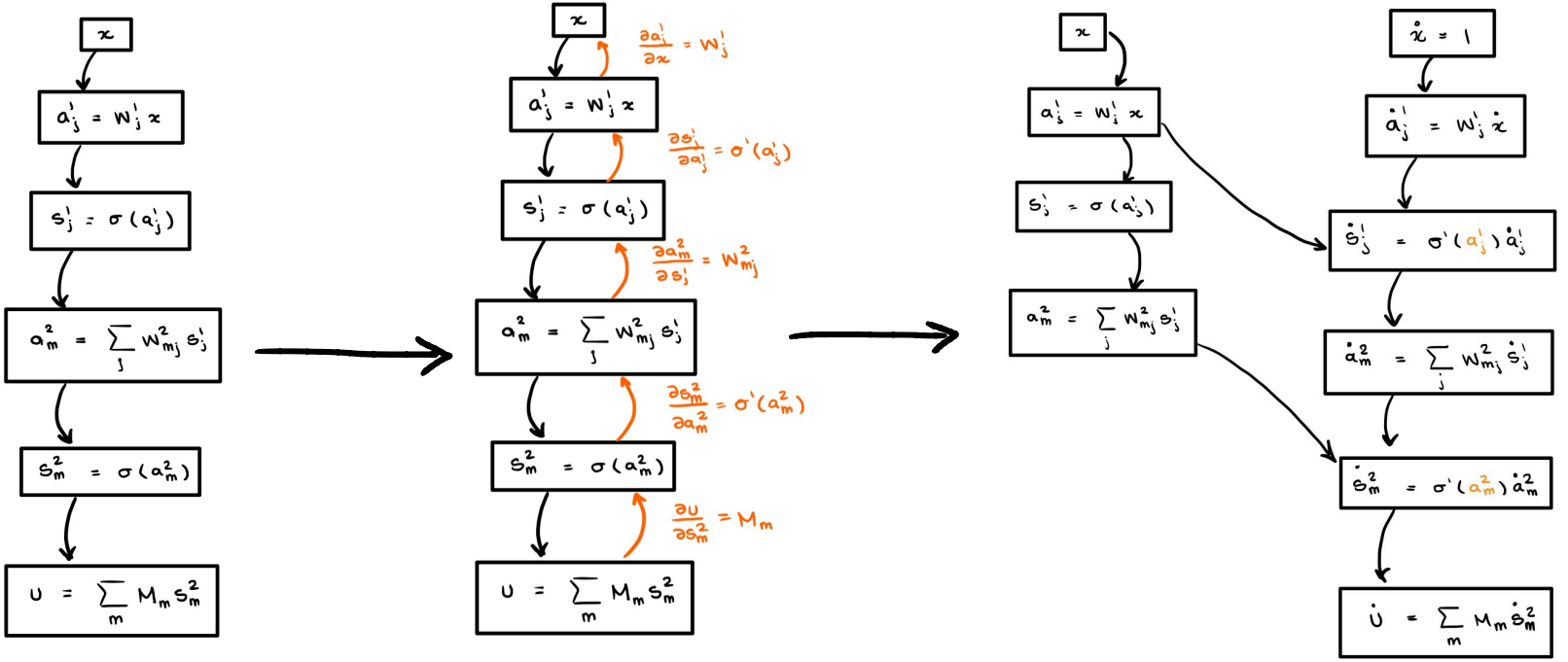
Autodiff Algorithm

- Forwards → Backwards → New Graph



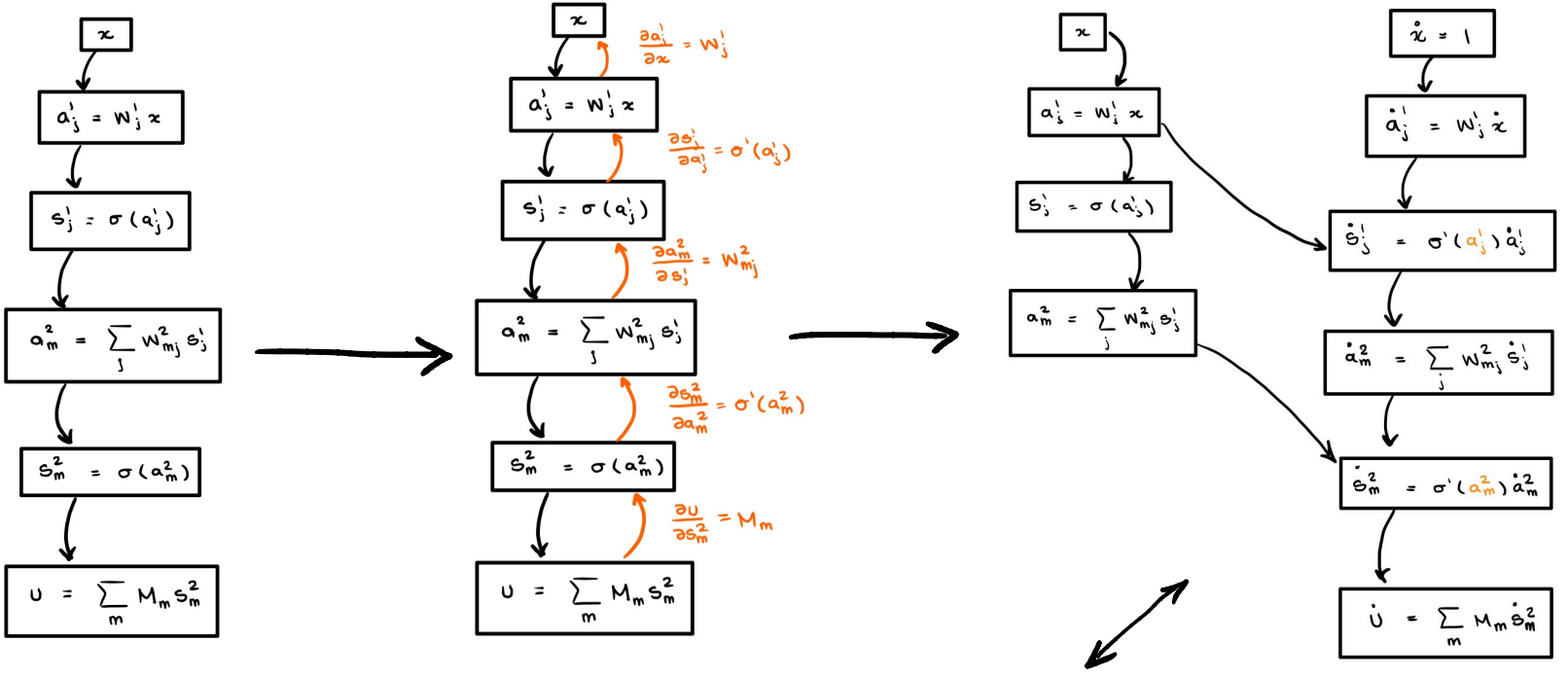
Autodiff Algorithm

- Forwards → Backwards → New Graph



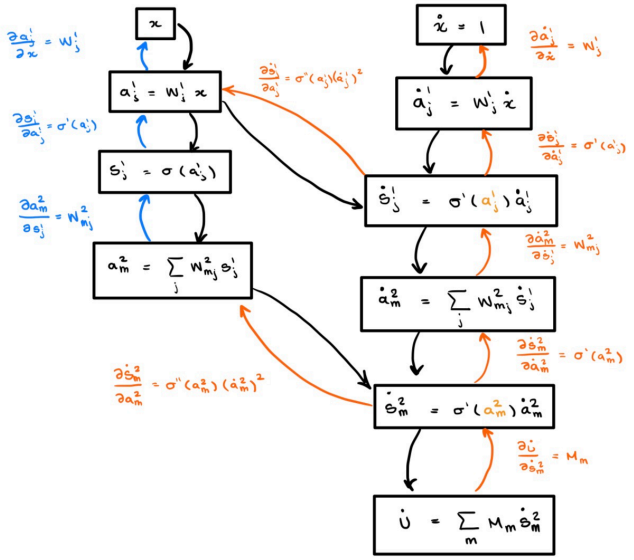
Autodiff Algorithm

- Forwards → Backwards → New Graph



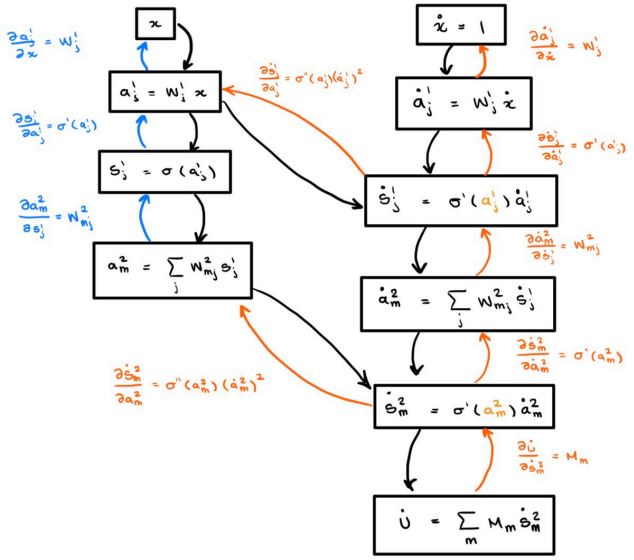
$$U' = M \cdot \sigma'(W^2 \cdot \sigma(W^1 x)) \cdot (W^2 \cdot \sigma'(W^1 x)) \cdot W^1$$

Computation is Repeated Through Chain Rule

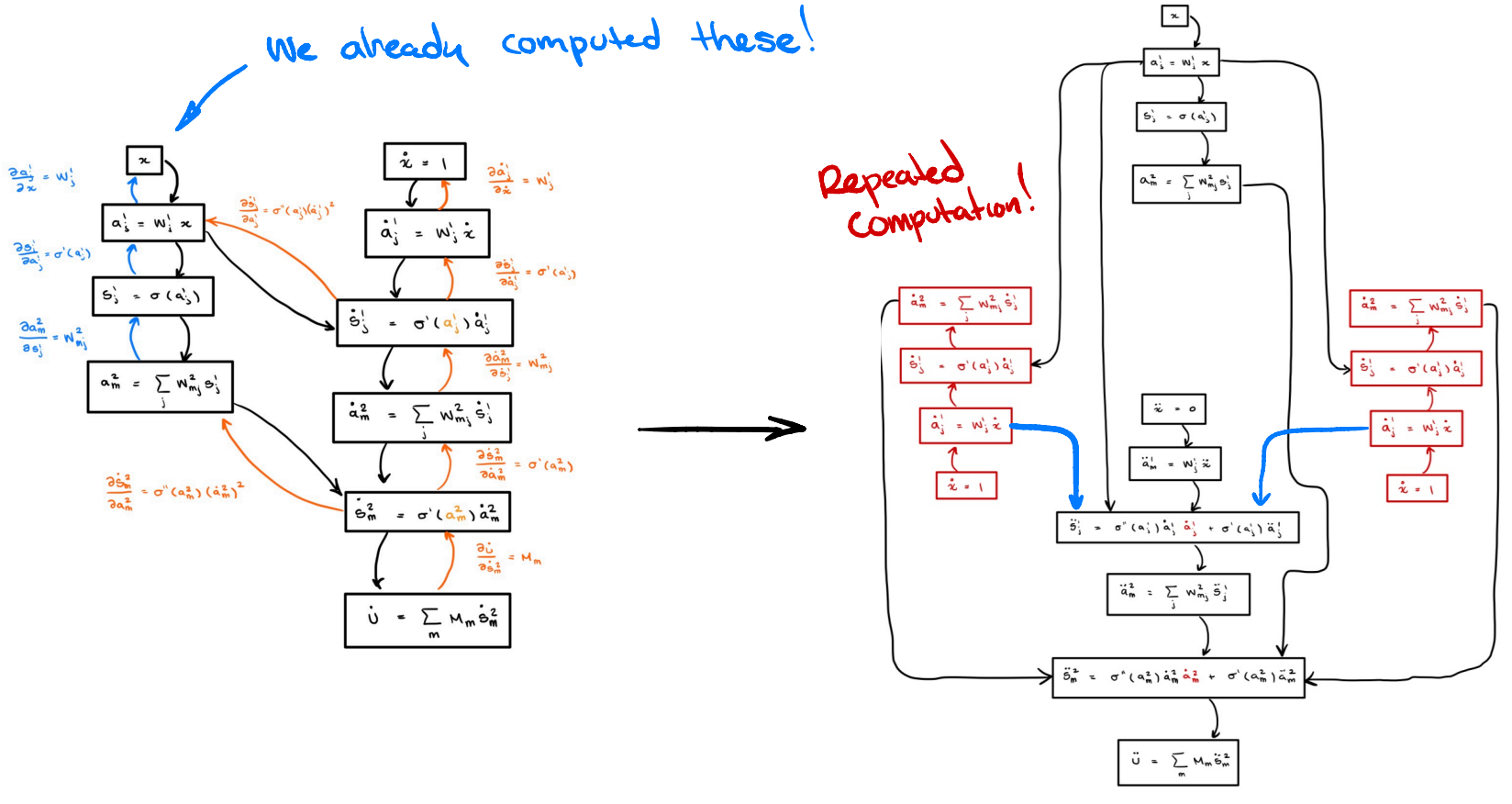


Computation is Repeated Through Chain Rule

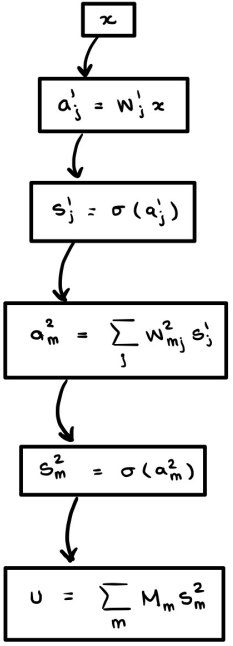
We already computed these!



Computation is Repeated Through Chain Rule

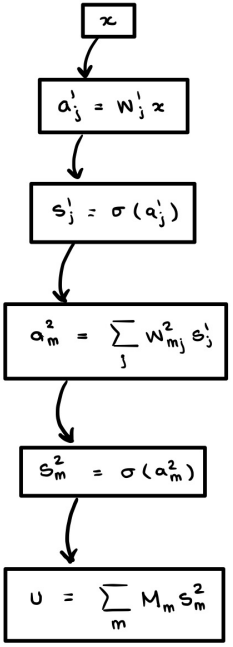


Autodiff Graph Grows Exponentially!

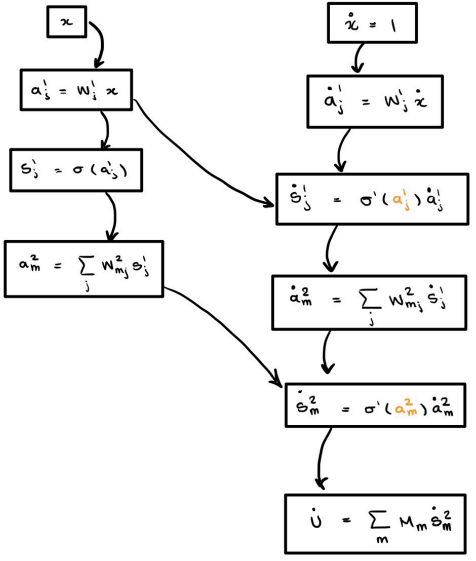


Zeroth

Autodiff Graph Grows Exponentially!

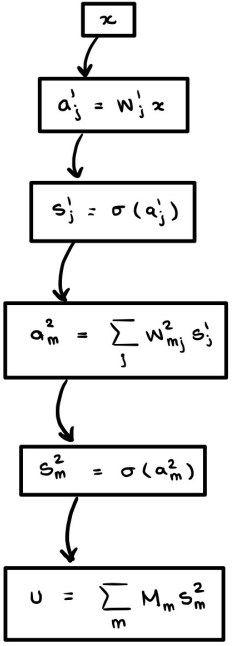


Zeroth

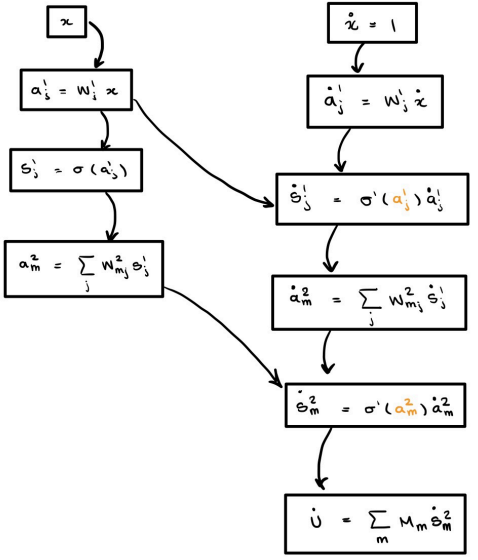


First

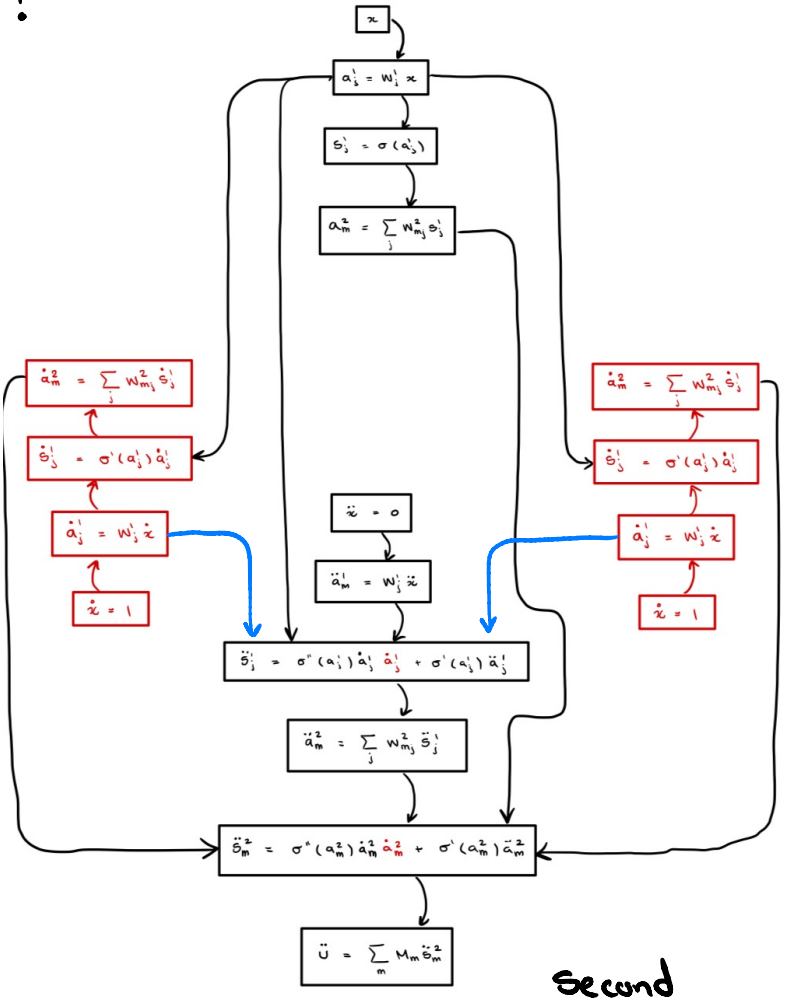
Autodiff Graph Grows Exponentially!



Zeroth

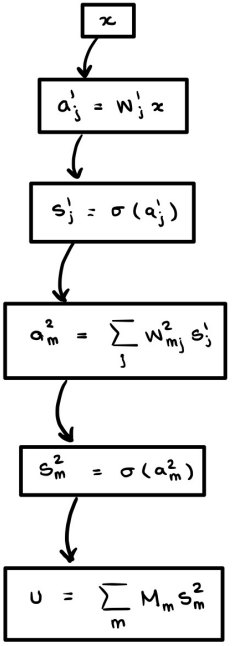


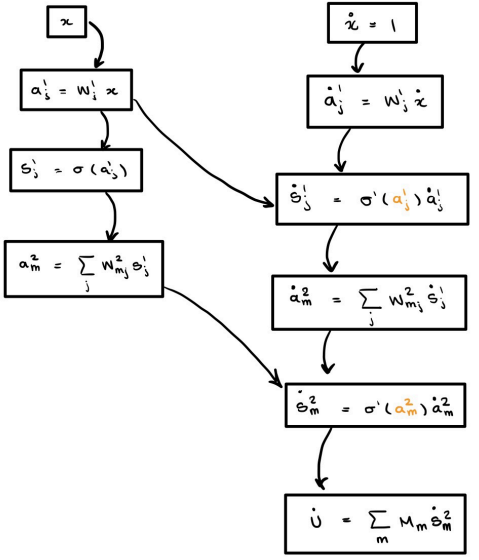
First



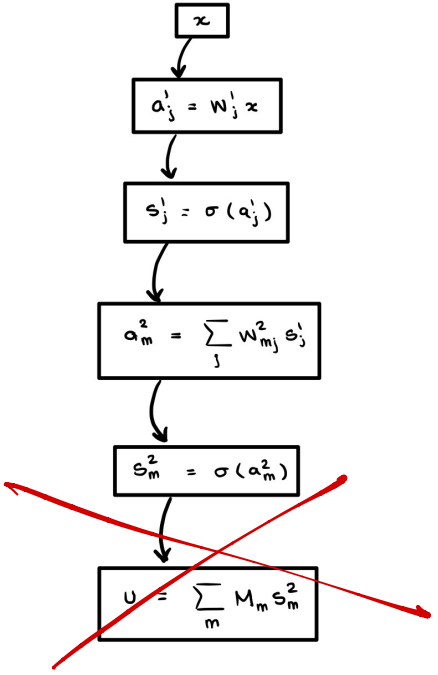
Second

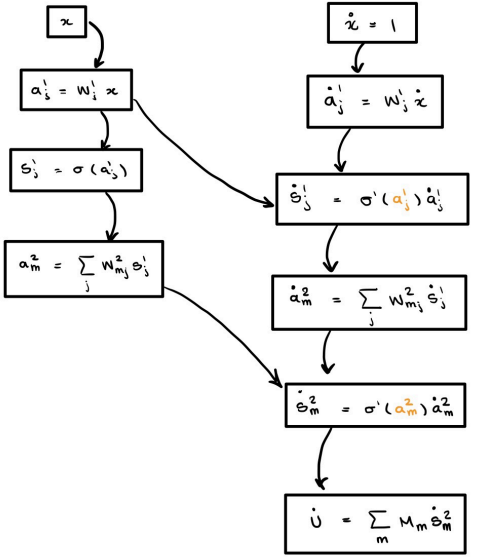
Counting Nodes I



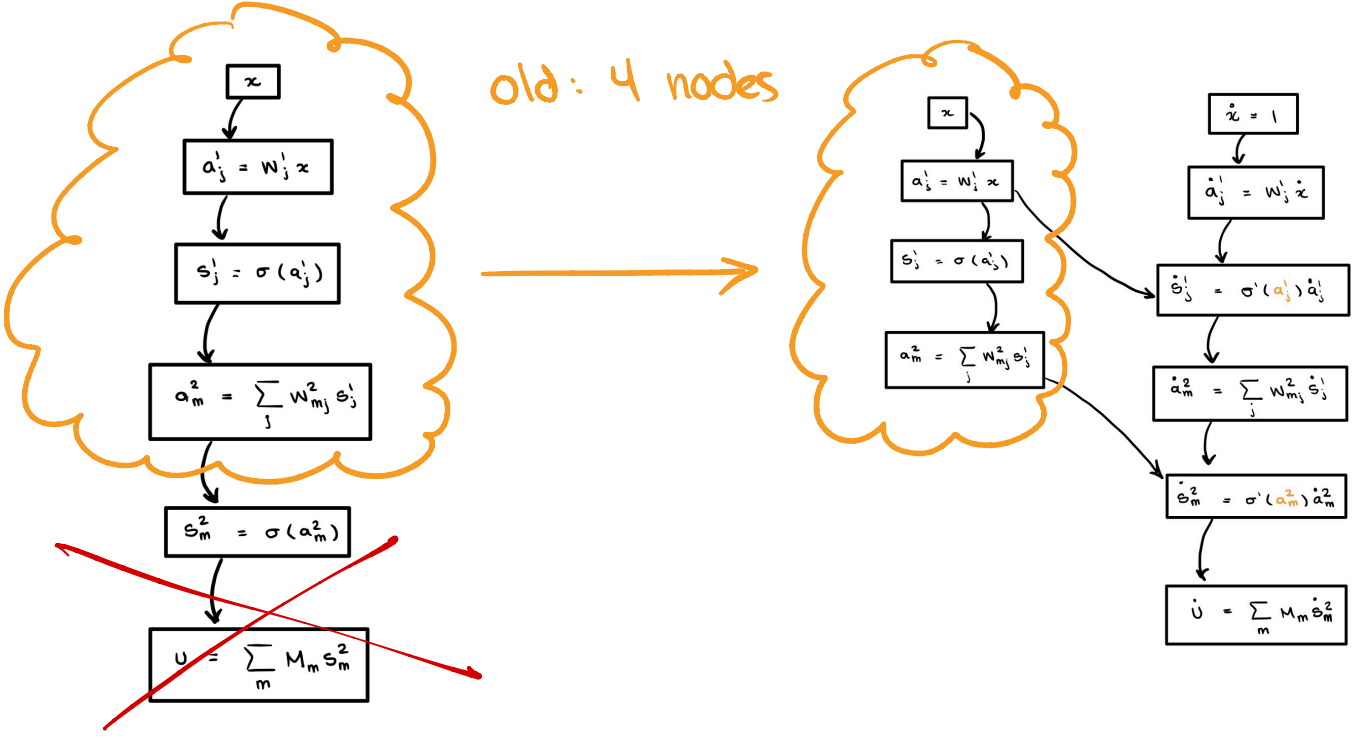


Counting Nodes I

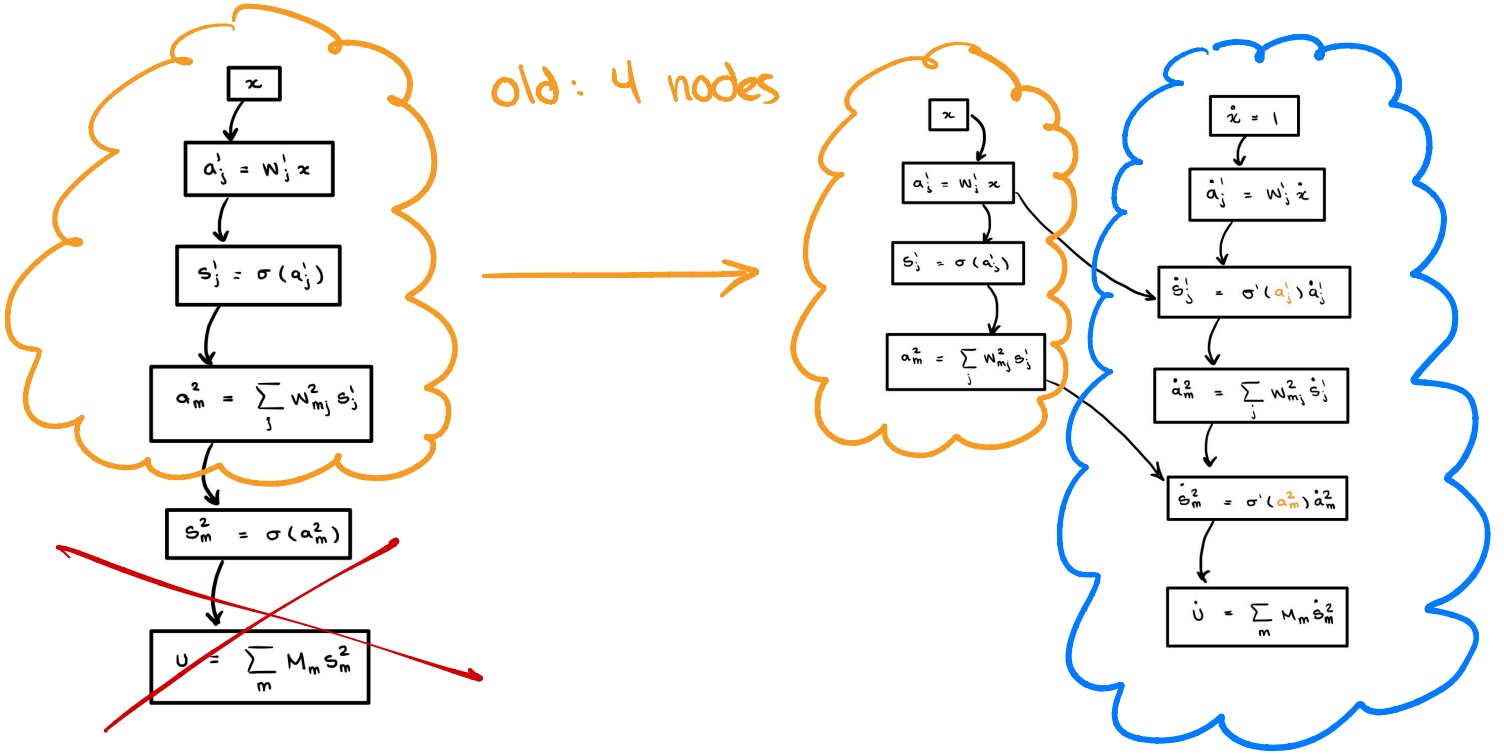




Counting Nodes I

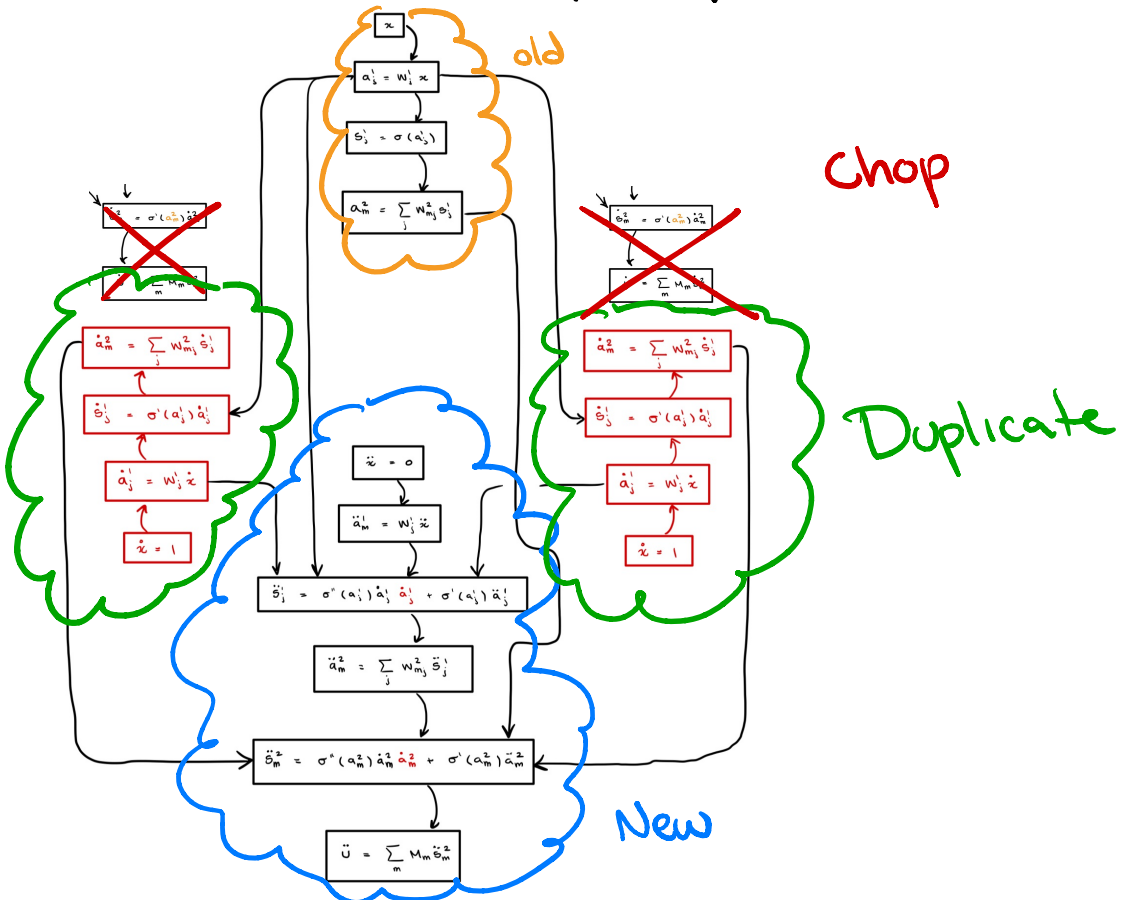


Counting Nodes I

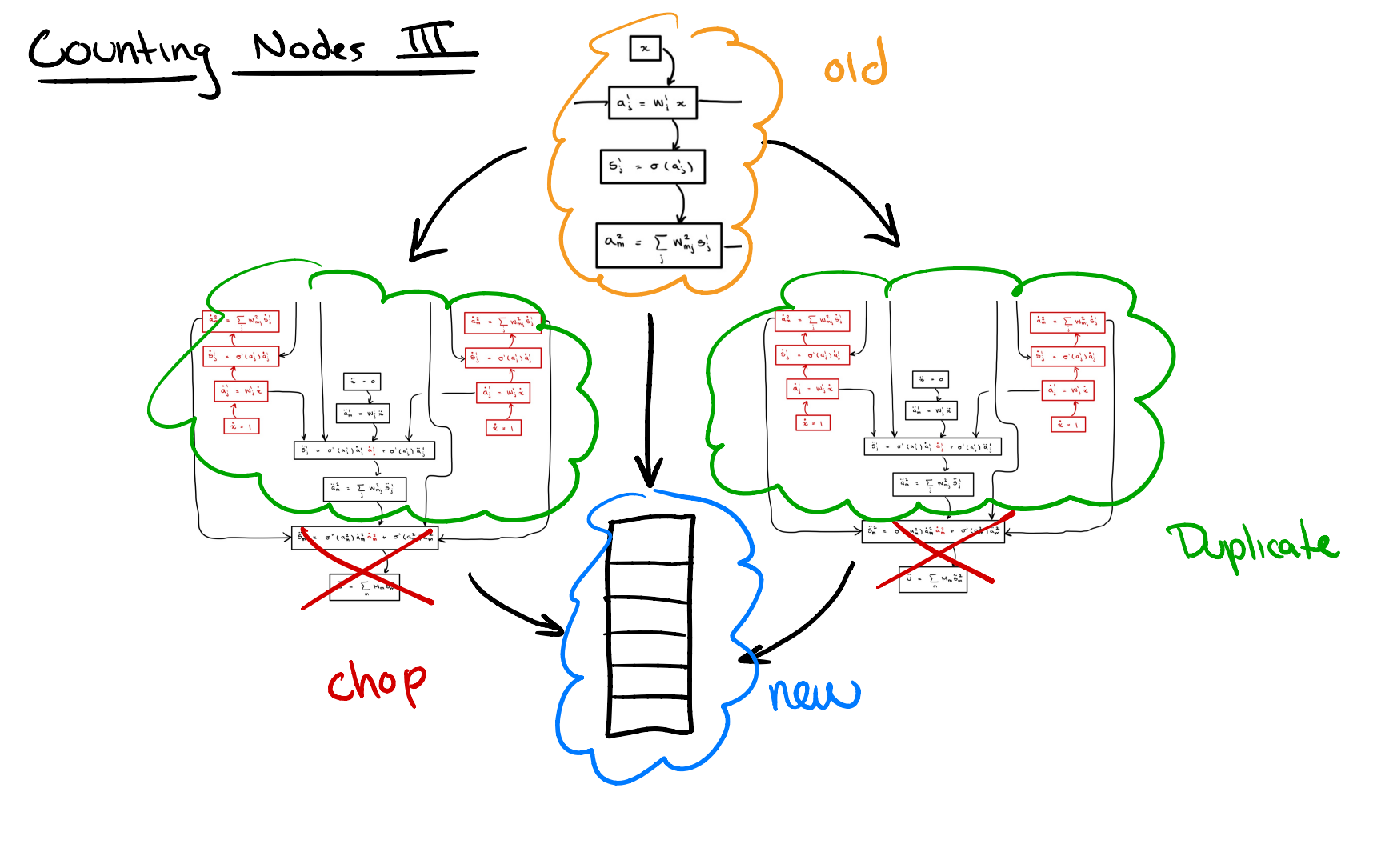


Counting Nodes II

- 6 new, 4 old, chop & duplicate the rest



Counting Nodes III



Counting Nodes III:

- # Nodes (n) = 10 + 2 × (# Nodes (n-1) - 6)
- # Nodes (0) = 6

Counting Nodes III:

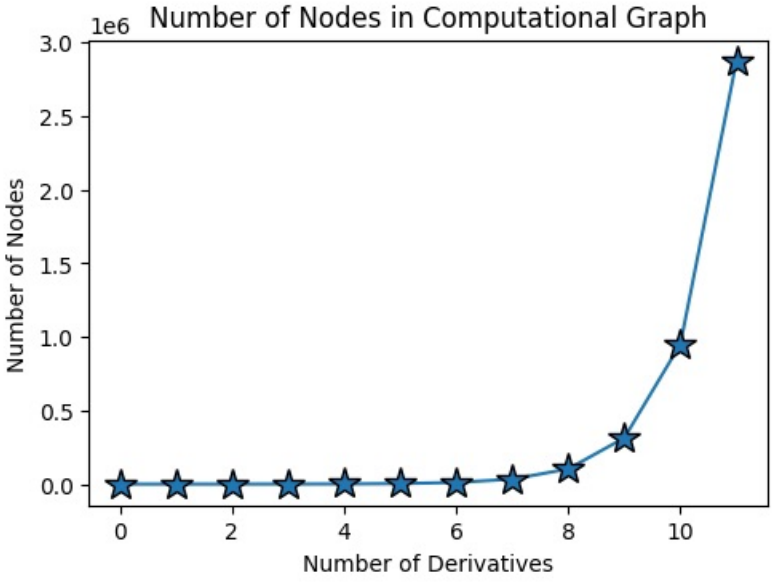
- $\#\text{Nodes}(n) = 10 + 2 \times (\#\text{Nodes}(n-1) - 6)$
- $\#\text{Nodes}(0) = 6$
- $\Rightarrow \#\text{Nodes}(n) = 2^{n+1} - 2$

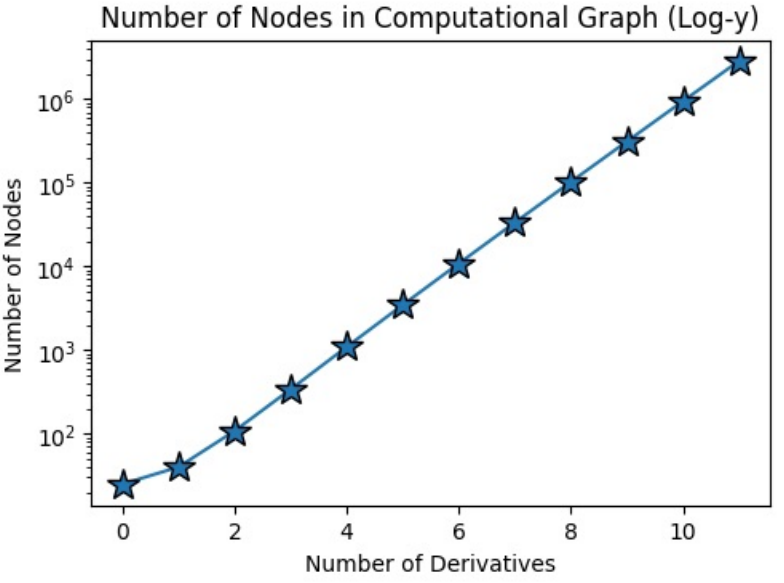
Counting Nodes III:

- $\#\text{Nodes}(n) = 10 + 2 \times (\#\text{Nodes}(n-1) - 6)$
- $\#\text{Nodes}(0) = 6$
- $\Rightarrow \#\text{Nodes}(n) = 2^{n+1} - 2$
- Worse if we have to evaluate <u>all</u> derivatives up to n

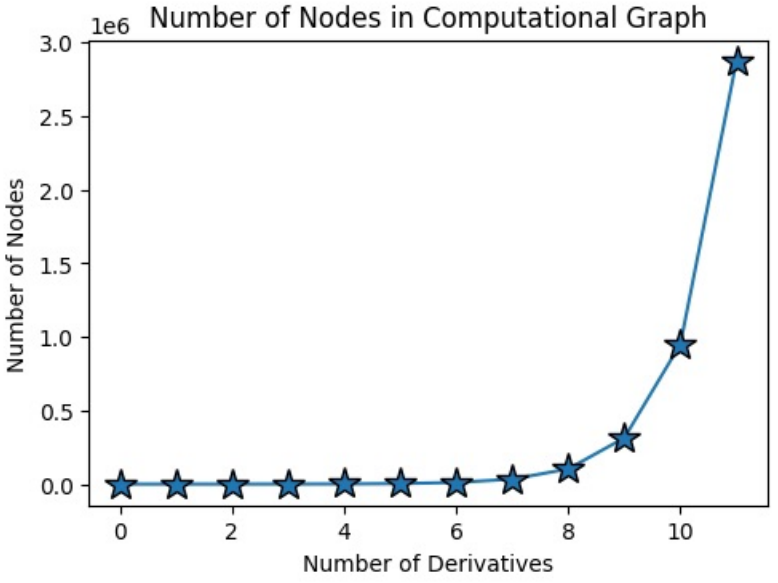
$$\sum_{k=0}^{n} \#\text{Nodes}(k) = 2^{n+2} - 2$$

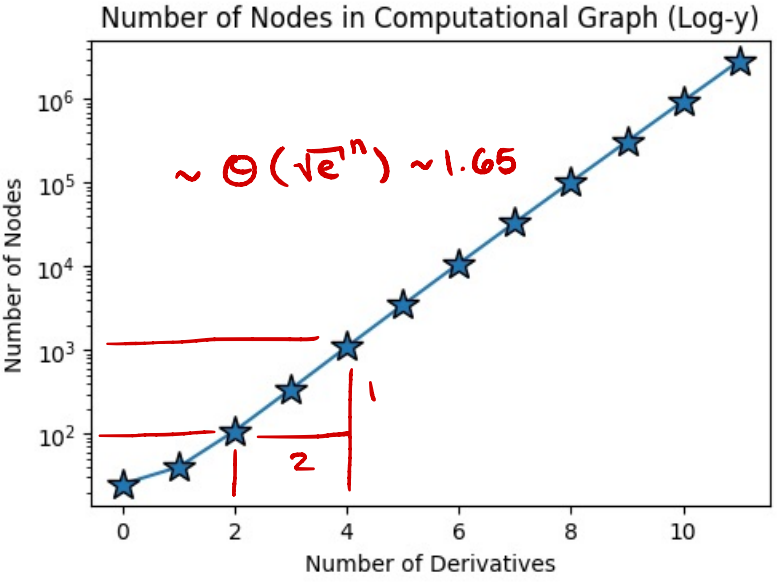
Counting Nodes V : Empirical



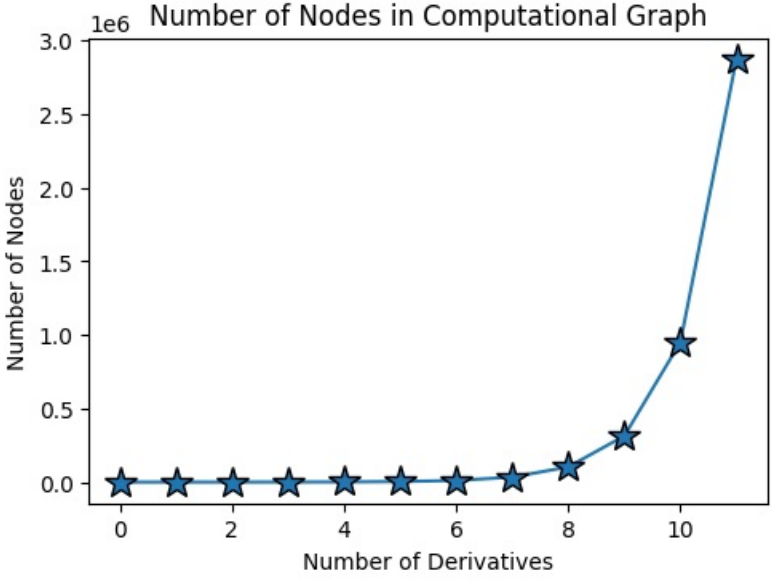


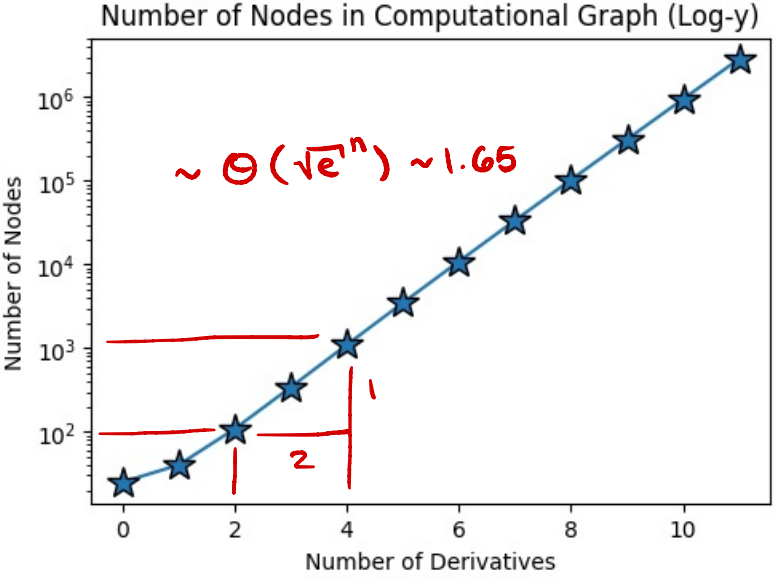
Counting Nodes V : Empirical





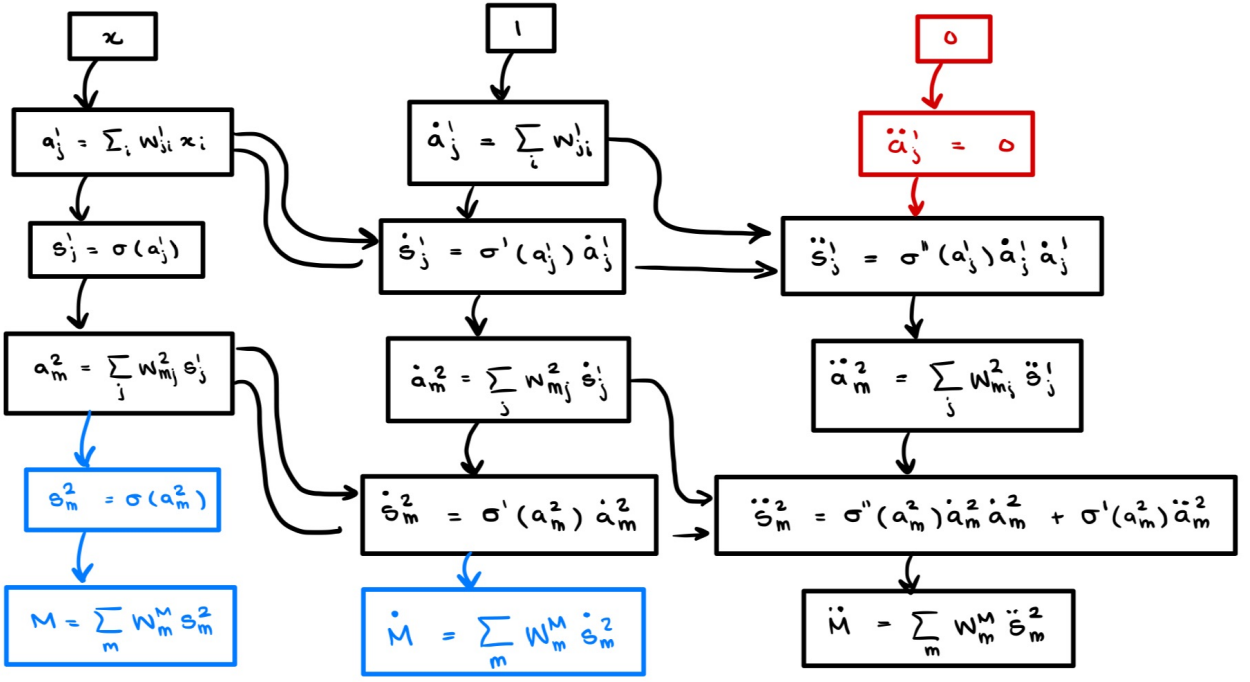
Counting Nodes V : Empirical



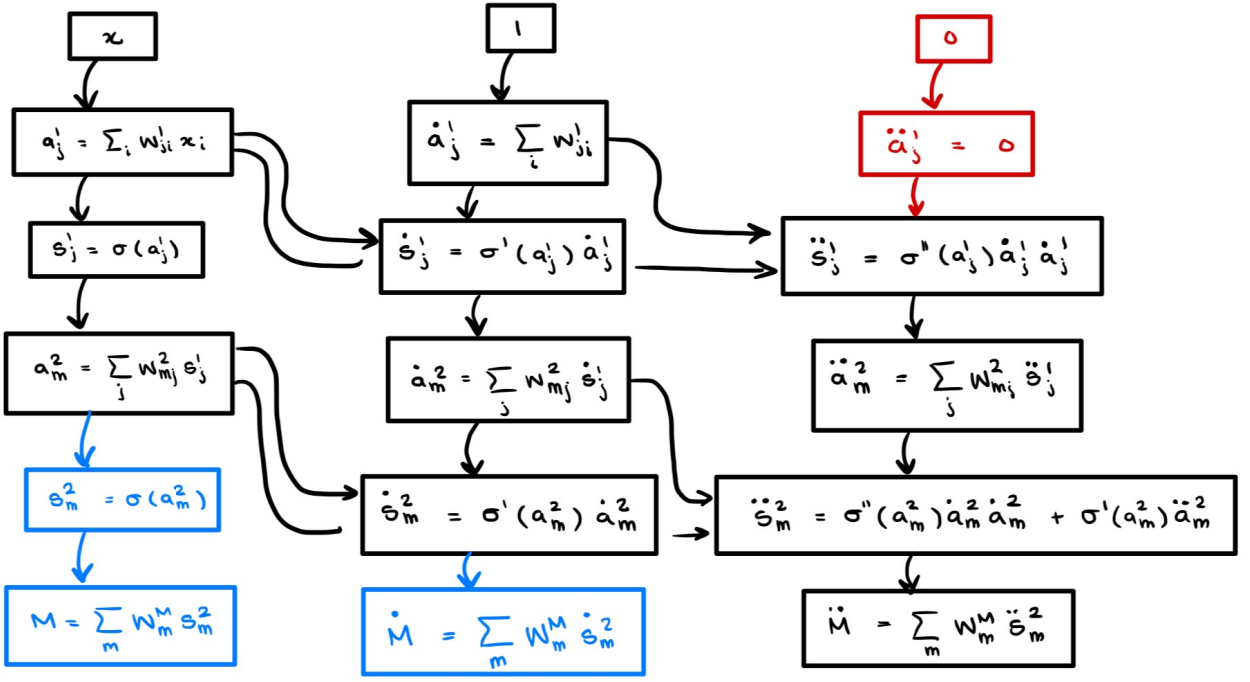


- Graph optimizations beat 2^n, but still exponential!

What Does the Minimal Graph for v^4 look like?



What Does the Minimal Graph for v^4 look like?



→ n-TangentProp will construct this instead!

TangentProp & n-TangentProp

Tangent Prop (Simard, Victorri, LeCun, Denker, 1991)

- Developed before Autodiff was in widespread use

Tangent Prop (Simard, Victorri, LeCun, Denker, 1991)

- Developed before Autodiff was in widespread use
- First order constraint improves training convergence (c.f. Sobolev Loss!)

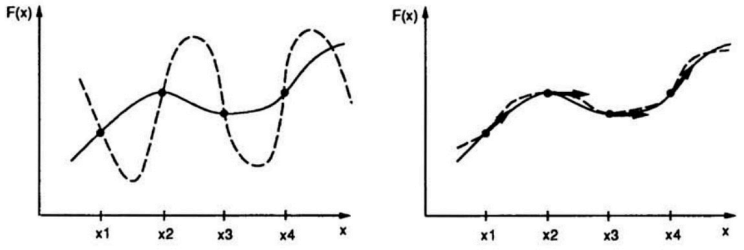


Figure 2: Learning a given function (solid line) from a limited set of example (x_1 to x_4). The fitted curves are shown in dotted line. Top: The only constraint is that the fitted curve goes through the examples. Bottom: The fitted curves not only goes through each examples but also its derivatives evaluated at the examples agree with the derivatives of the given function.

From SVLCD 1991

Tangent Prop (Simard, Victorri, LeCun, Denker, 1991)

- Developed **before** Autodiff was in widespread use
- First order constraint improves training convergence (c.f. Sobolev Loss!)

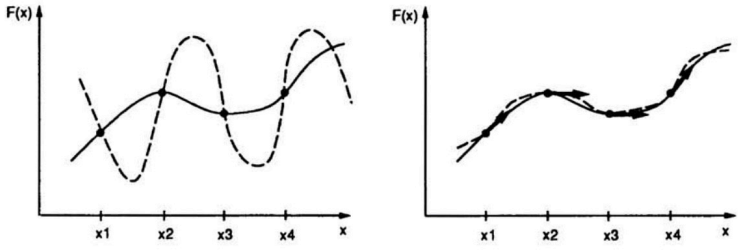


Figure 2: Learning a given function (solid line) from a limited set of example (x_1 to x_4). The fitted curves are shown in dotted line. Top: The only constraint is that the fitted curve goes through the examples. Bottom: The fitted curves not only goes through each examples but also its derivatives evaluated at the examples agree with the derivatives of the given function.

From SVLCD 1991

Forward

- $a_i^l = \sum_j w_{ij}^l x_j^{l-1}$, $x_j^l = \sigma(a_j^l)$

Derivative

- $\dot{a}_i^l = \sum_j w_{ij}^l z_j^{l-1}$, $z_j^l = \sigma'(a_j^l)\dot{a}_j^l$

Tangent Prop (Simard, Victorri, LeCun, Denker, 1991)

- Developed <u>before</u> Autodiff was in widespread use
- First order constraint improves training convergence (c.f. Sobolev Loss!)

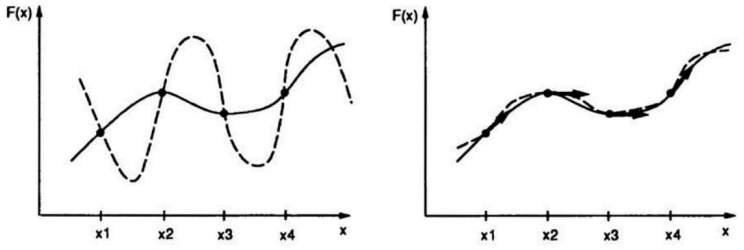


Figure 2: Learning a given function (solid line) from a limited set of example (x_1 to x_4). The fitted curves are shown in dotted line. Top: The only constraint is that the fitted curve goes through the examples. Bottom: The fitted curves not only goes through each examples but also its derivatives evaluated at the examples agree with the derivatives of the given function.

From SVLCD 1991

Forward

- $a_i^l = \sum_j w_{ij}^l x_j^{l-1}$, $x_j^l = \sigma(a_j^l)$

Derivative

- $\dot{a}_i^l = \sum_j w_{ij}^l z_j^{l-1}$, $z_j^l = \sigma'(a_j^l)\dot{a}_j^l$

- Both computed during forward

n-Tangent Prop I

- Leibniz formula: Product Rule

$$\frac{d^n}{dx^n}(fg) = \sum_{k=0}^{n} C^L(k,n) f^{(k)} g^{(n-k)}$$

n-Tangent Prop I

- Leibniz formula: Product Rule

$$\frac{d^n}{dx^n}(fg) = \sum_{k=0}^{n} C^L(k,n) f^{(k)} g^{(n-k)}$$

- Faà di Bruno formula: Chain Rule

$$\frac{d^n}{dx^n}(f \circ g) = \sum_{\bar{p} \in P(n)} C^F_{\bar{p},n} f^{(|\bar{p}|)} \circ g \prod_{j=1}^{n} \left(g^{(j)}\right)^{p_j}$$

n-TangentProp II

- Use Faà to compute all derivs in a single forward pass

n - TangentProp II

- Use Faà to compute all derivs in a single Forward pass

- Replace $\dot{a}_i^l = \sum_j w_{ij}^l z_j^{l-1}$, $z_j^l = \sigma'(a_j^l)\dot{a}_j^l$ with

$$\partial^m a_i^l = \sum_j w_{ij}^l \partial^m z_j^{l-1},$$

$$\partial^m z_j^l = \sum_{\bar{P} \in P(m)} C^F_{\bar{P},m} \sigma^{(|\bar{P}|)}(a_j^l) \prod_{\nu=1}^{m} \left(\partial^\nu a_j^l\right)^{\bar{P}_\nu}$$

n - TangentProp II

- Use Faà to compute all derivs in a single Forward pass

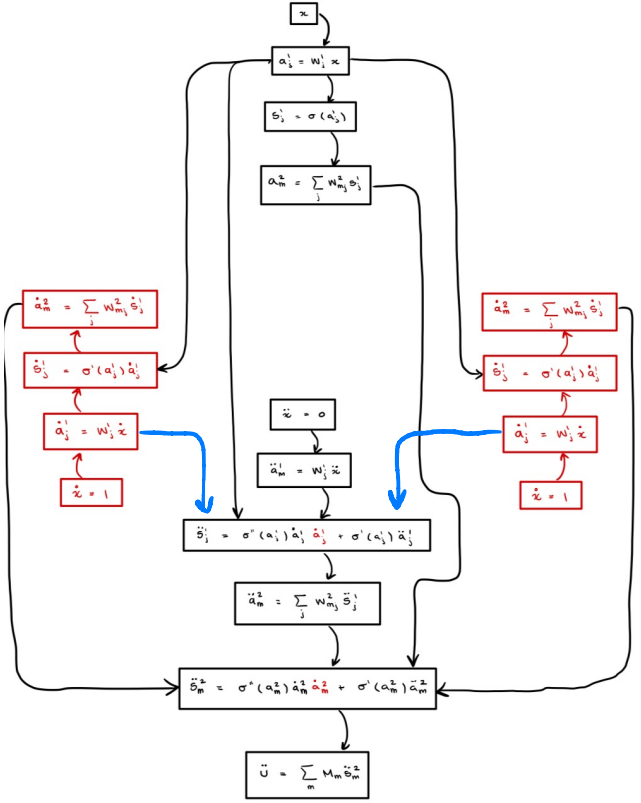
- Replace $\dot{a}_i^l = \sum_j w_{ij}^l z_j^{l-1}$, $z_j^l = \sigma'(a_j^l)\dot{a}_j^l$ with

$$\partial^m a_i^l = \sum_j w_{ij}^l \partial^m z_j^{l-1},$$

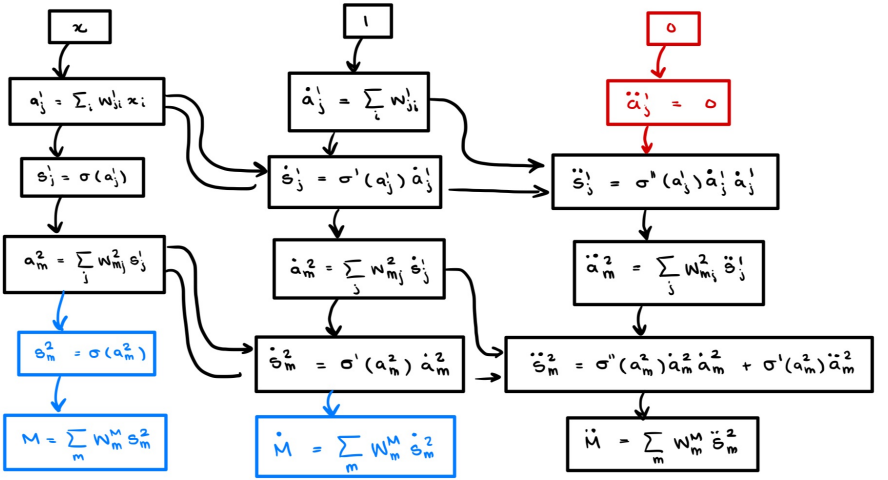
$$\partial^m z_j^l = \sum_{\bar{p} \in P(m)} C^F_{\bar{p},m} \sigma^{(|\bar{p}|)}(a_j^l) \prod_{\nu=1}^{m} (\partial^\nu a_j^l)^{\bar{p}_\nu}$$

* Forces us to compute derivatives in order!

Computational Graphs

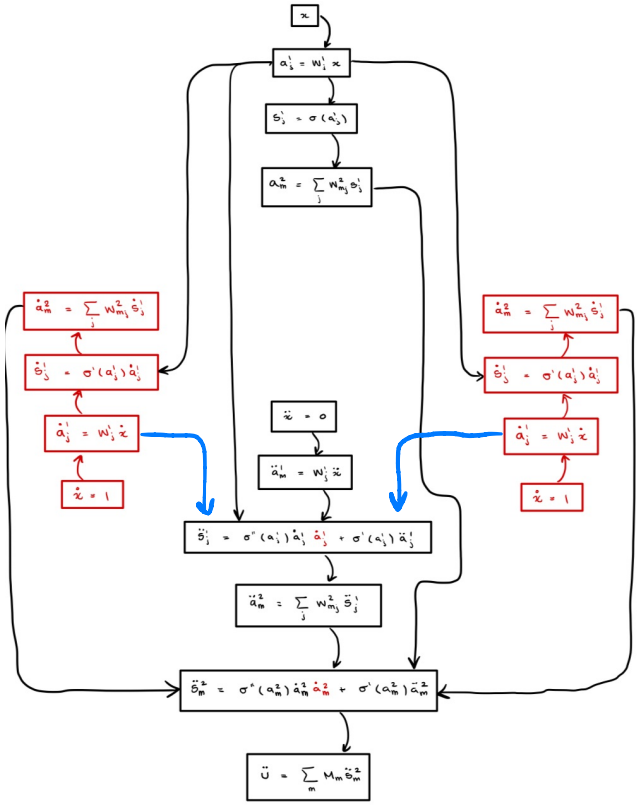


Autodiff

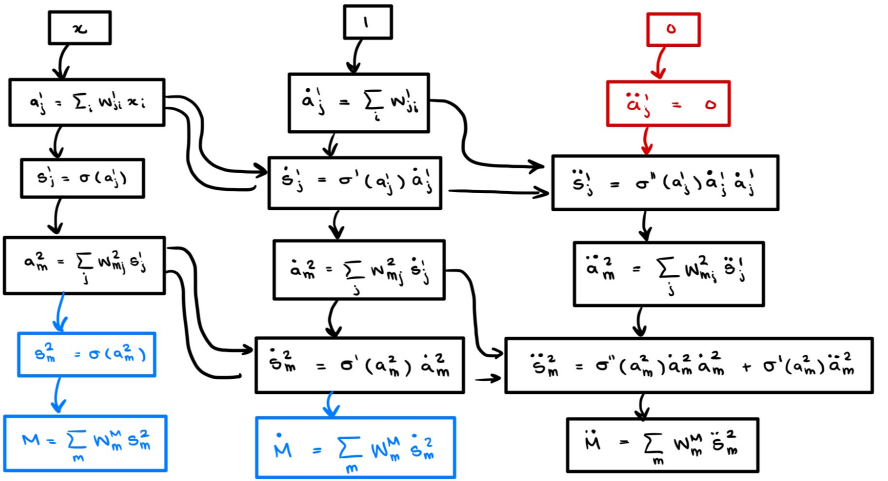


n-Tangent Prop

Computational Graphs



Autodiff



n-Tangent Prop

To compute 0, 1, 2 derivs:

AD: 18 + 10 + 6 = 34 nodes

nTP: 18 nodes

Runtime Analysis

- Linear # of nodes in computation graph

Runtime Analysis

- Linear # of nodes in computation graph
- Assume original computation time is $\Theta(C)$, then

Ad: $\Theta(2^n C)$, exponential

n-TP: $\Theta(nC)$, linear!

Runtime Analysis

- Linear # of nodes in computation graph
- Assume original computation time is $\Theta(C)$, then

Ad: $\Theta(2^n C)$, n-TP: $\Theta(nC)$

exponential

linear!

- Ignores a lot: We will do empirical analysis below

Pedantic Runtime Analysis I

- Faà di Bruno introduces a non-trivial dependence on $|P(n)|$.

Pedantic Runtime Analysis I

- Faà di Bruno introduces a non-trivial dependence on $|P(n)|$.
- Hardy - Ramanujan 1917: $|P(n)| = \Theta\left(\frac{1}{n} e^{\sqrt{n}}\right)$

Pedantic Runtime Analysis I

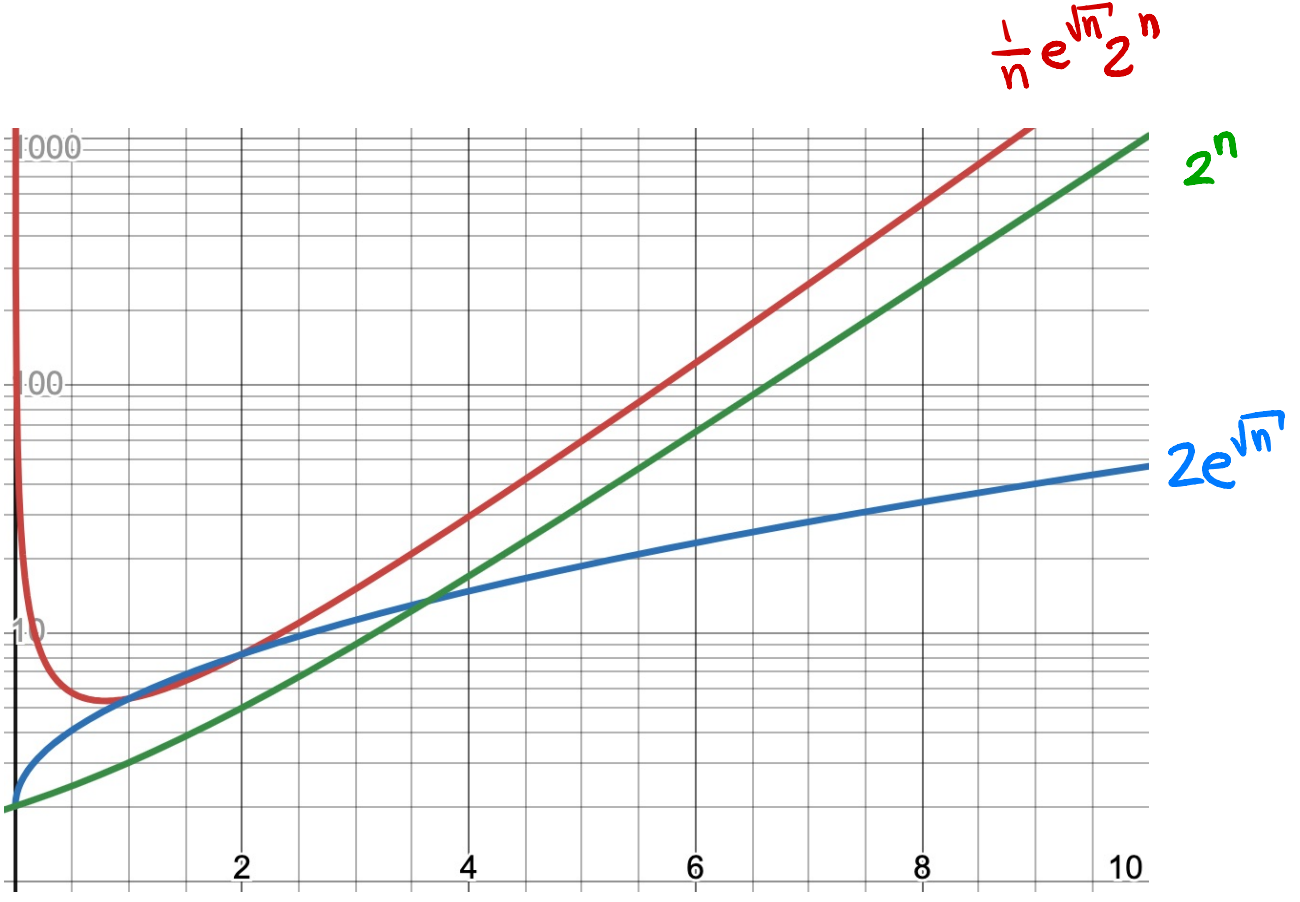
- Faà di Bruno introduces a non-trivial dependence on $|P(n)|$.
- Hardy-Ramanujan 1917: $|P(n)| = \Theta\left(\frac{1}{n} e^{\sqrt{n}}\right)$
- But this will appear in both n-TangentProp & Autodiff!

Pedantic Runtime Analysis I

- Faà di Bruno introduces a non-trivial dependence on $|P(n)|$.
- Hardy-Ramanujan 1917: $|P(n)| = \Theta\left(\frac{1}{n} e^{\sqrt{n}}\right)$
- But this will appear in both n-TangentProp & Autodiff!

$$\text{AD} = \Theta\left(\frac{e^{\sqrt{n}}}{n} 9^n\right), \quad \text{nTP} = \Theta\left(e^{\sqrt{n}}\right)$$

Pedantic Runtime Analysis II



Results

Results: Neural Network I

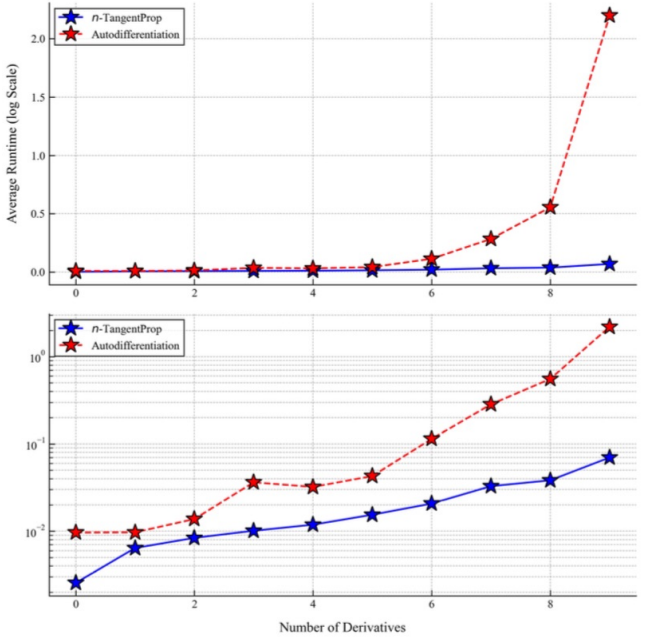


Fig. 1. Average runtime for a combined forward and backwards pass using autodifferentiation (red) and n-TangentProp (blue). The top and bottom frames show the same data, however the bottom frame is plotted with a logarithmic y-axis. Each model is run 100 times and the average for each trial is plotted. 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.

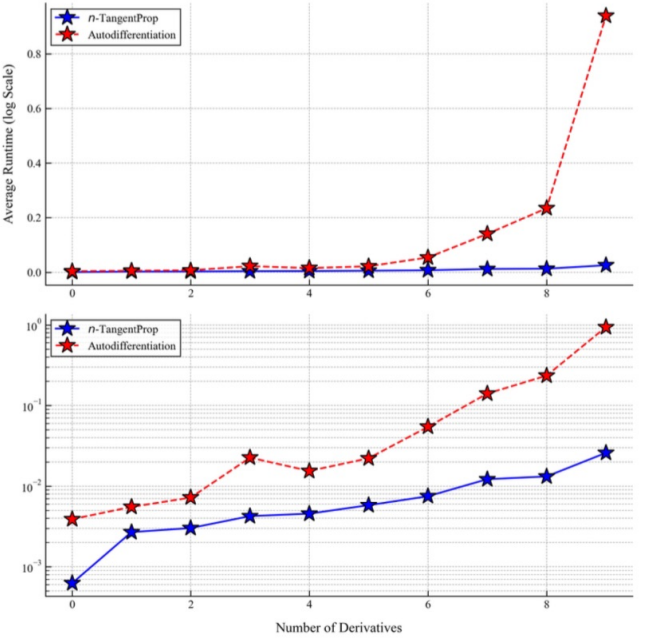
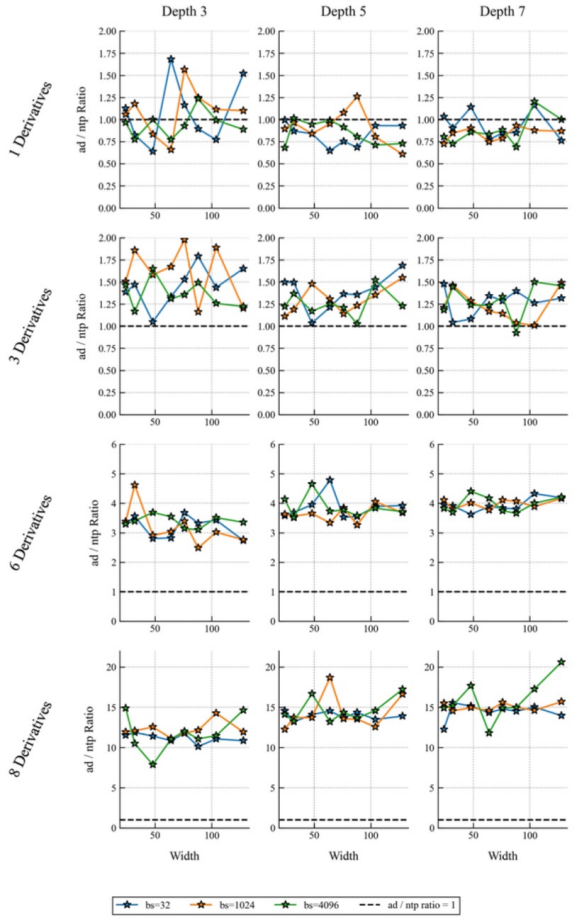


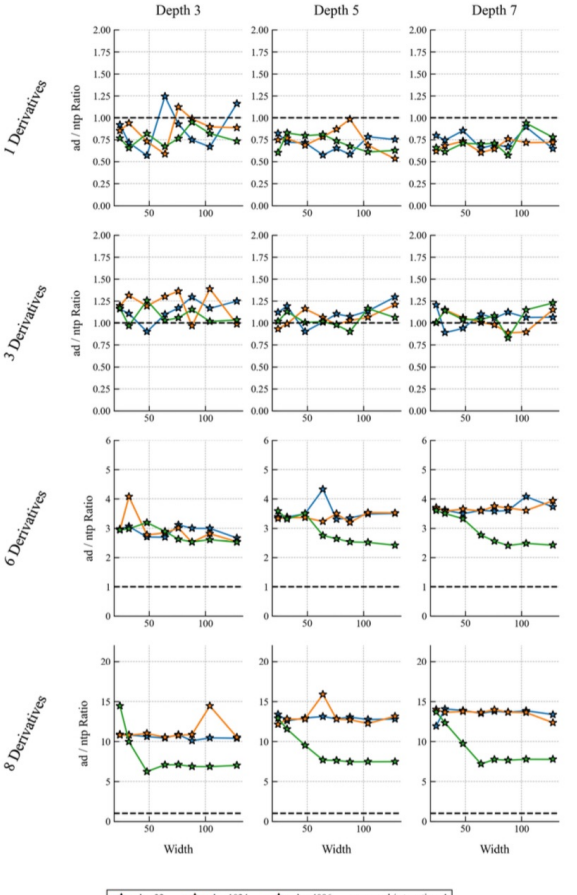
Fig. 2. Forward pass times for the model shown in Figure 1. The top and bottom frames show the same data, however the bottom frame is plotted with a logarithmic y-axis. Each model is run 100 times and the average for each trial is plotted. The network has 3 hidden layers of 24 neurons each, a common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

Results: Neural Networks II

Forward



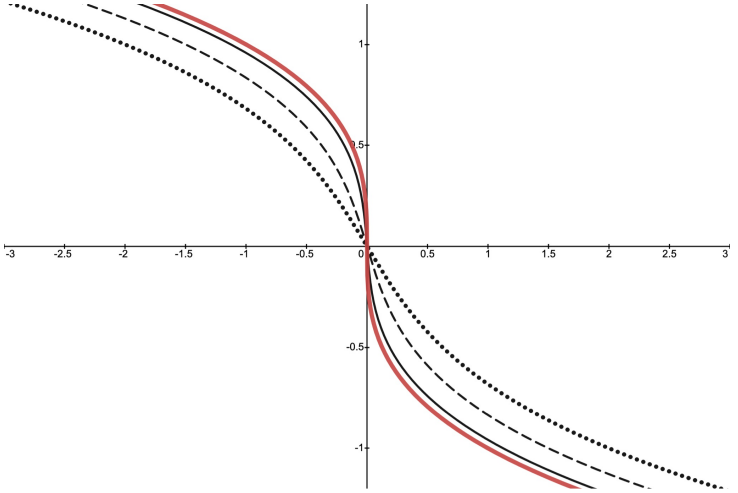
Combined



Self-Similar Burgers I

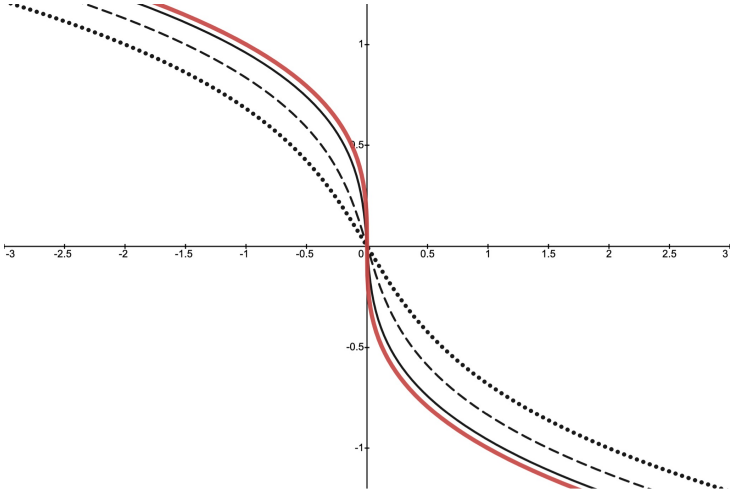
- Burgers: $\partial_t u + u \partial_x u = 0$

Self-Similar Burgers I



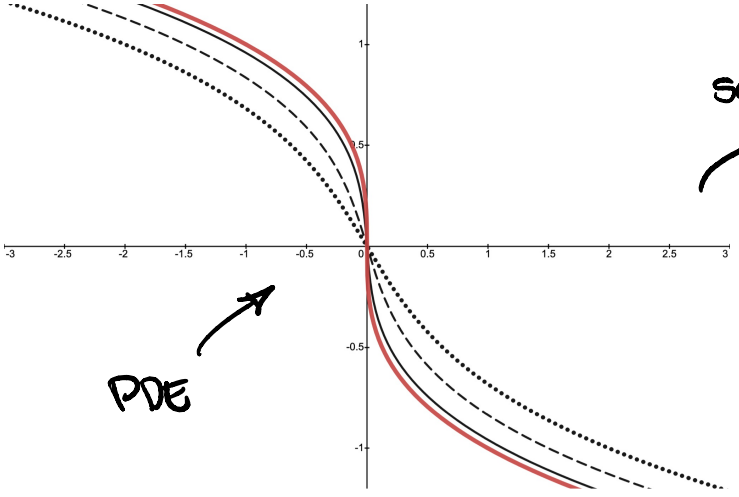
- Burgers: $\partial_t u + u \partial_x u = 0$
- Shocks: $|\partial_x u| \to \infty$ in finite time

Self-Similar Burgers I

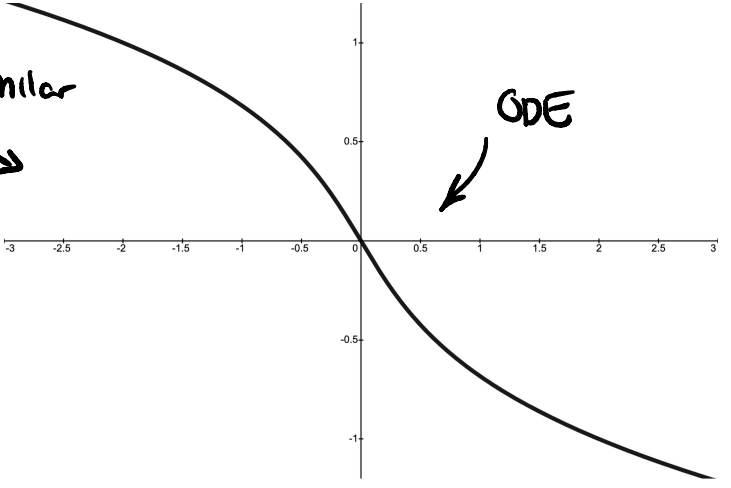


- Burgers: $\partial_t u + u \partial_x u = 0$
- Shocks: $|\partial_x u| \to \infty$ in finite time
- This makes analysis hard

Self-Similar Burgers I



self-similar



- Burgers: $\partial_t u + u \partial_x u = 0$
- Shocks: $|\partial_x u| \to \infty$ in finite time
- This makes analysis hard

Self-Similar Burgers II

- Self-Similar transform takes advantage of Power law relation between spatial & temporal scales

Self-Similar Burgess II

- Self-Similar transform takes advantage of Power law relation between spatial & temporal scales

- $X = \dfrac{x}{(1-t)^{1+\lambda}}$, $U(x,t) = (1-t)^{\lambda} W(X)$

$$\Rightarrow -\lambda W + \left((1+\lambda)X + W\right)W' = 0$$

Self-Similar Burgess II

- Self-Similar transform takes advantage of Power law relation between spatial & temporal scales

- $X = \dfrac{x}{(1-t)^{1+\lambda}}$, $U(x,t) = (1-t)^{\lambda} W(X)$

$$\Rightarrow -\lambda W + \left((1+\lambda)X + W\right)W' = 0$$

- ODE in X, Allows us to study solution <u>AT</u> the shock, since X^{-1} captures the shock formation dynamics

Self-Similar Burgess II

- Self-Similar transform takes advantage of Power law relation between spatial & temporal scales

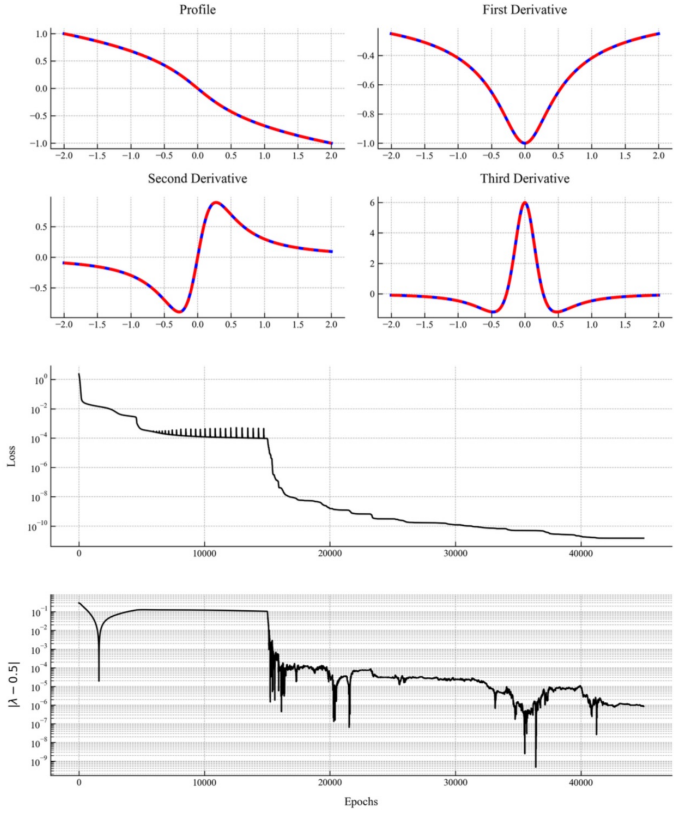
- $X = \frac{x}{(1-t)^{1+\lambda}}$, $U(x,t) = (1-t)^{\lambda} W(X)$

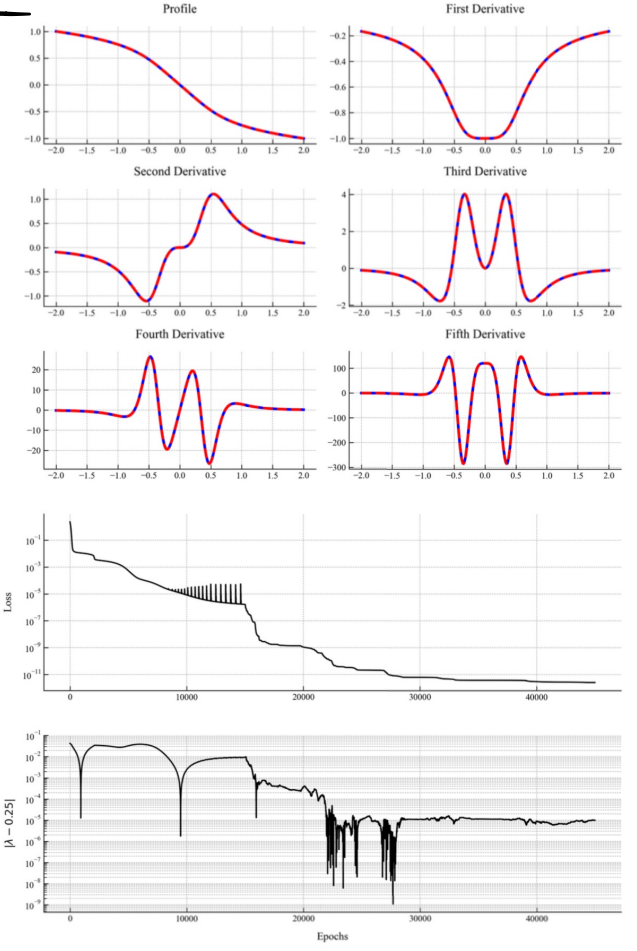
$$\Rightarrow -\lambda W + ((1+\lambda)X + W)W' = 0$$

- ODE in X, Allows us to study solution <u>AT</u> the shock, since X^{-1} captures the shock formation dynamics

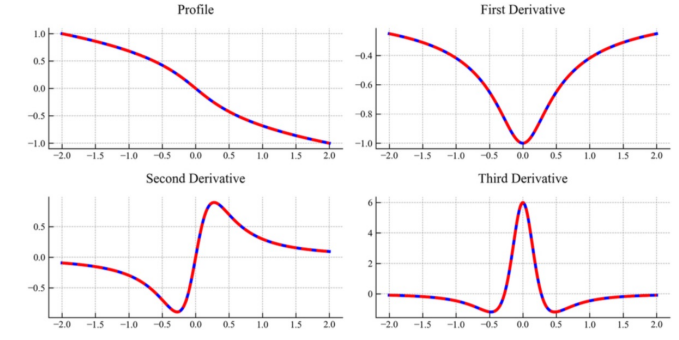
- PINN: Solve for W, λ <u>simultaneously</u>

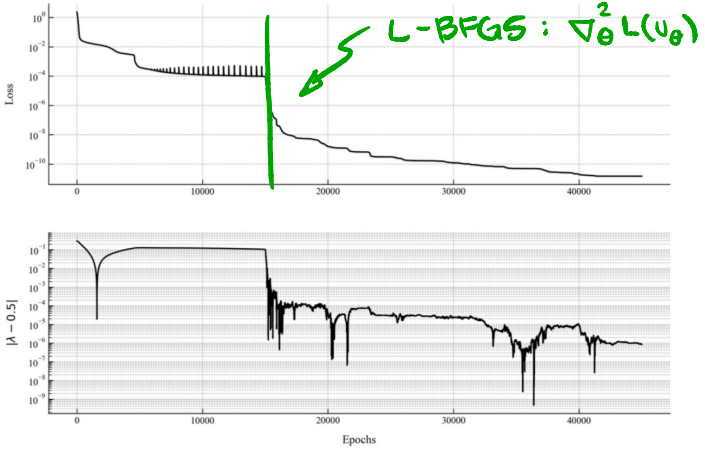
Results: Self-Similar Burgers I

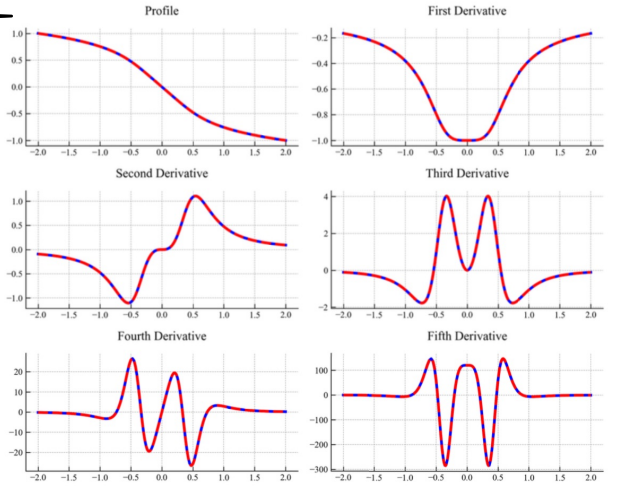


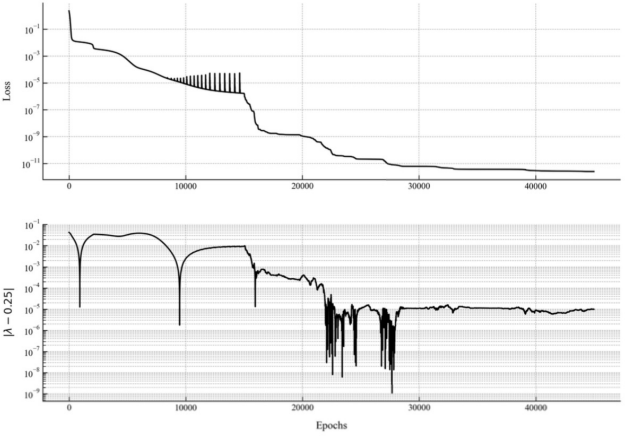


Results: Self-Similar Burgers I

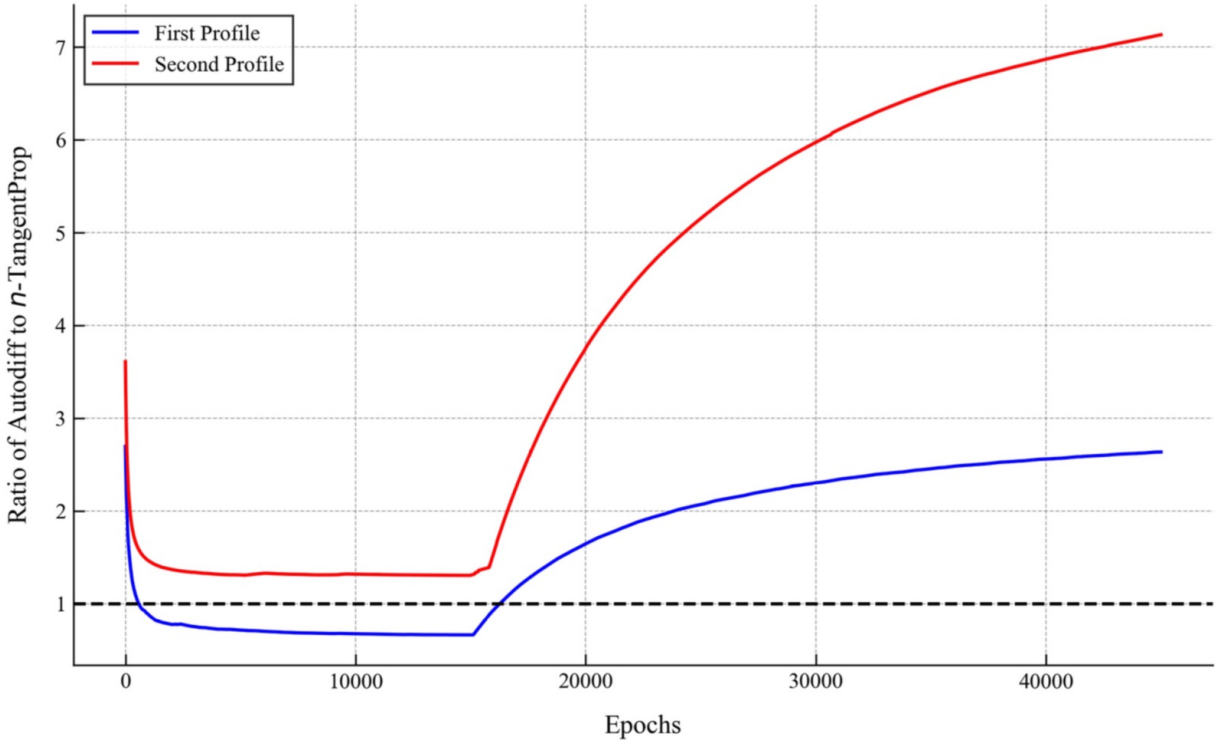




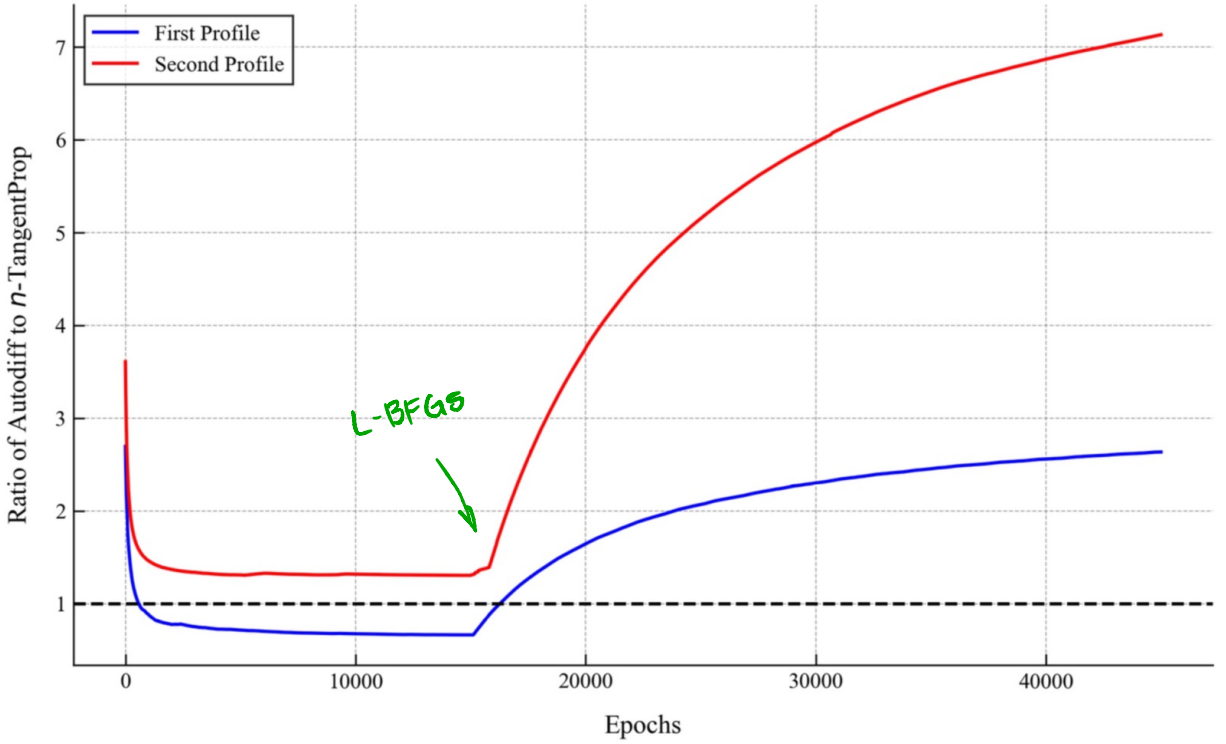




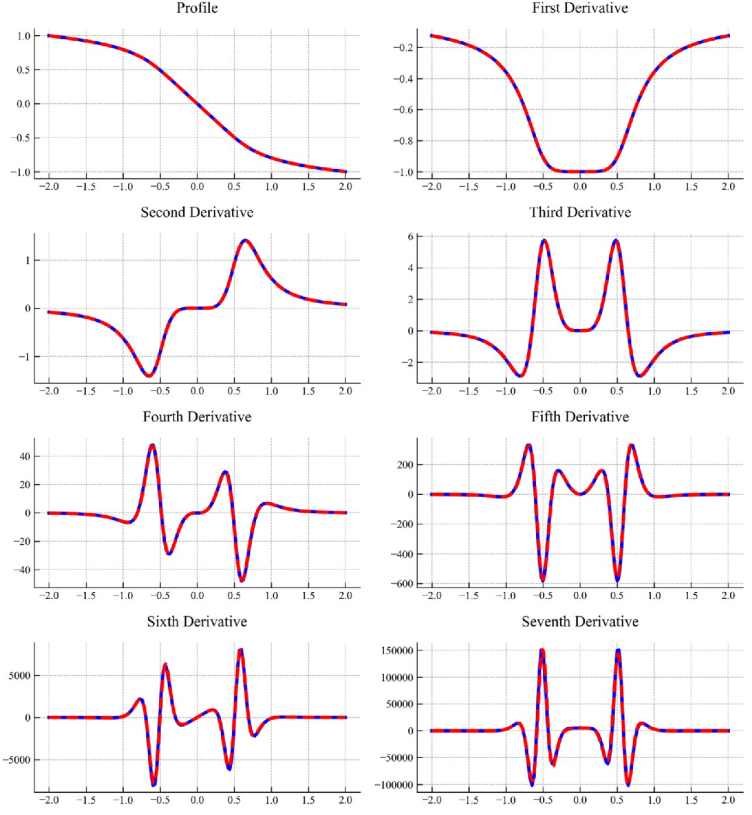
Results: Self-Similar Burgers II

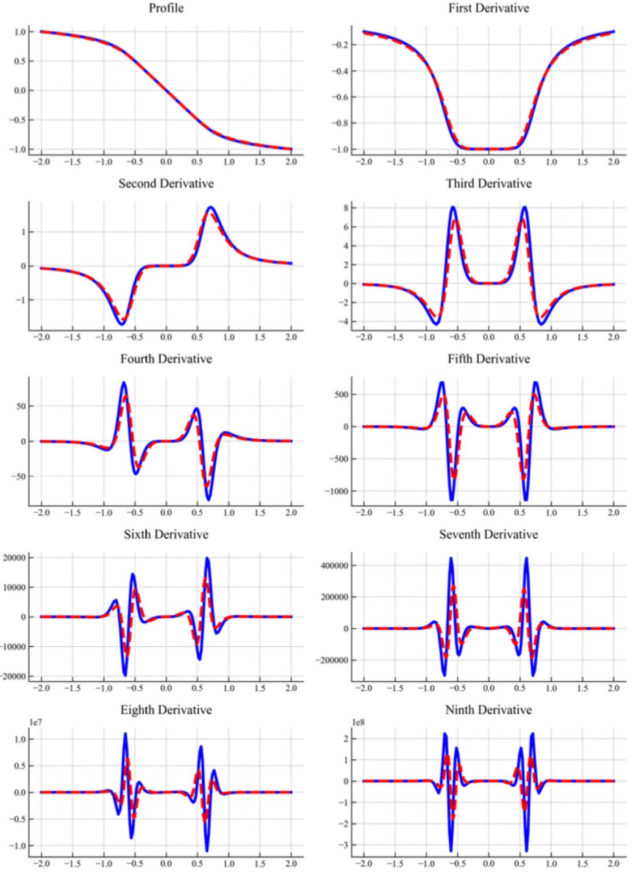


Results: Self-Similar Burgers II



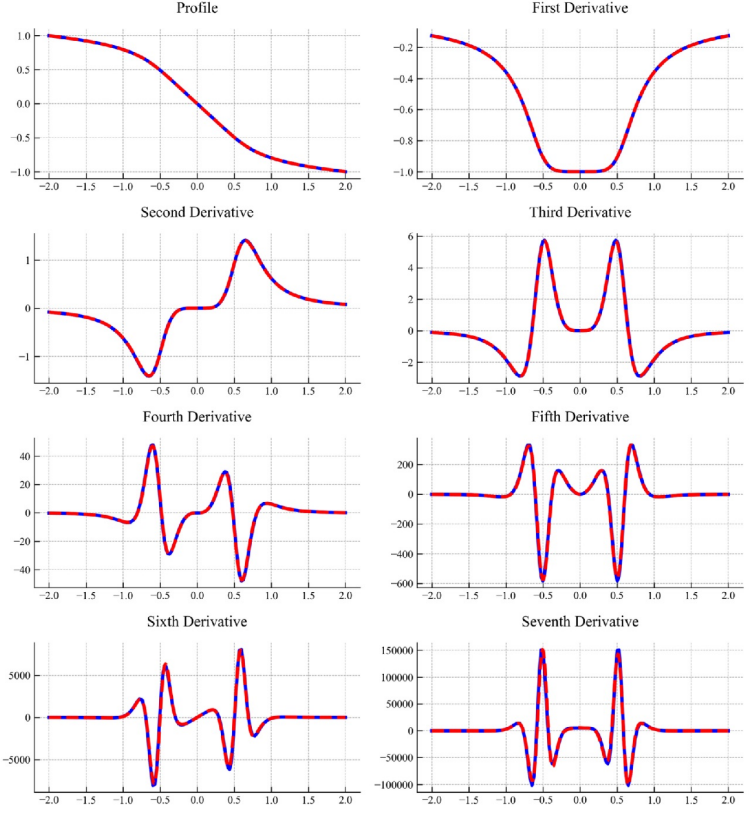
Results : Self-Similar Burgess III

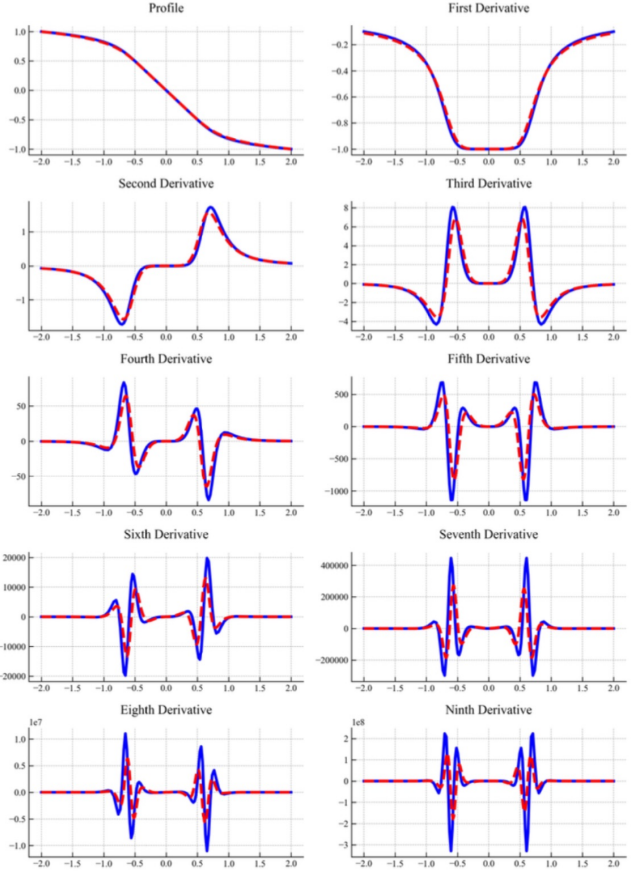




Results : Self-Similar Burgess III

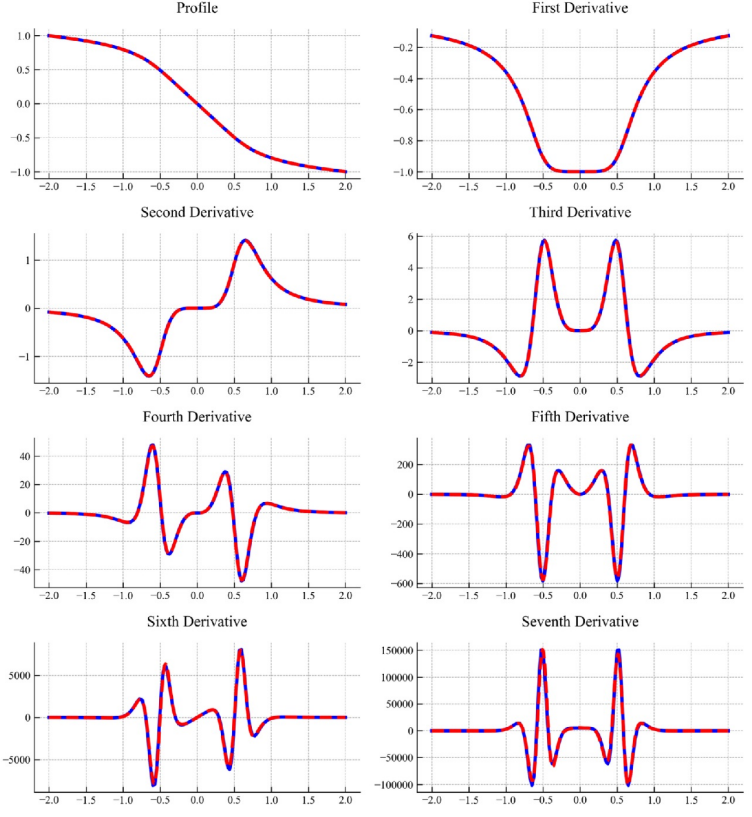
26x Faster



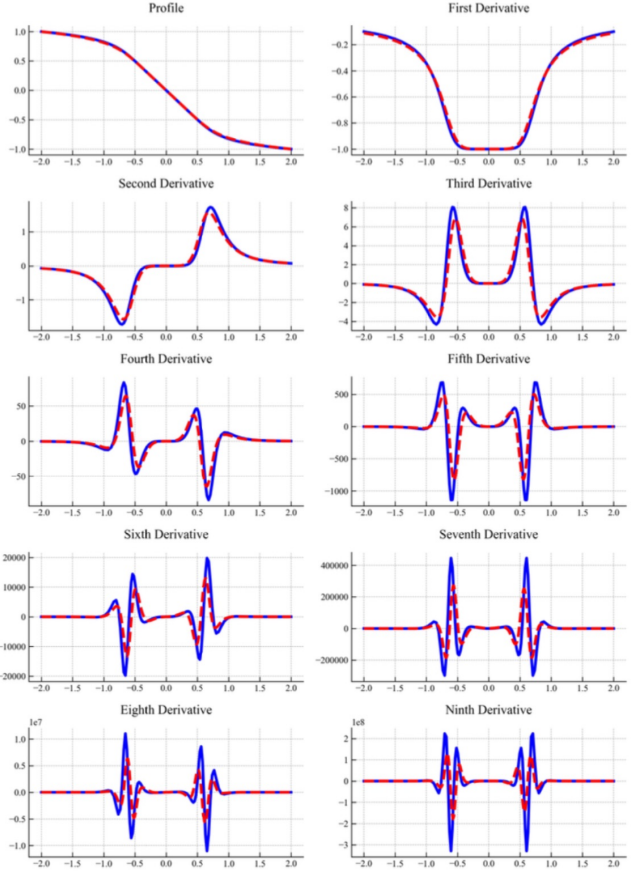


Results : Self-Similar Burgess III

26x Faster



65x Faster



Conclusions

Conclusions

- Use Sobolev Loss + n-TP for more efficient training

Conclusions

- Use Sobolev Loss + n-TP for more efficient training
- Faster training, less compute

Conclusions

- Use Sobolev Loss + n-TP for more efficient training
- Faster training, less compute
- Democratization of PINN training

Conclusions

- Use Sobolev Loss + n-TP for more efficient training
- Faster training, less compute
- Democratization of PINN training
- Multiprop, other applications?

Review

- Neural Networks → PINN

Review

- Neural Networks → PINN
- Autodiff is exponential in # of derivatives

Review

- Neural Networks → PINN
- Autodiff is exponential in # of derivatives
- n-TangentProp computes minimal comp. graph

Review

- Neural Networks → PINN
- Autodiff is exponential in # of derivatives
- n-Tangent Prop computes minimal comp. graph
- More accurate solutions, ~3x faster for simple problem, 65+ x faster on harder problem.

Thank You!