Learning Feynman integrals from differential equations with neural networks

Simone Zoia

with **Francesco Calisto**, **Ryan Moodie** (arXiv:2312.02067)

Loops and Legs, 16th April 2024

Feynman integrals are important, really

Essential ingredients of perturbative computations → particle phenomenology

Also: gravitational waves, cosmology, statistical mechanics, mathematics…

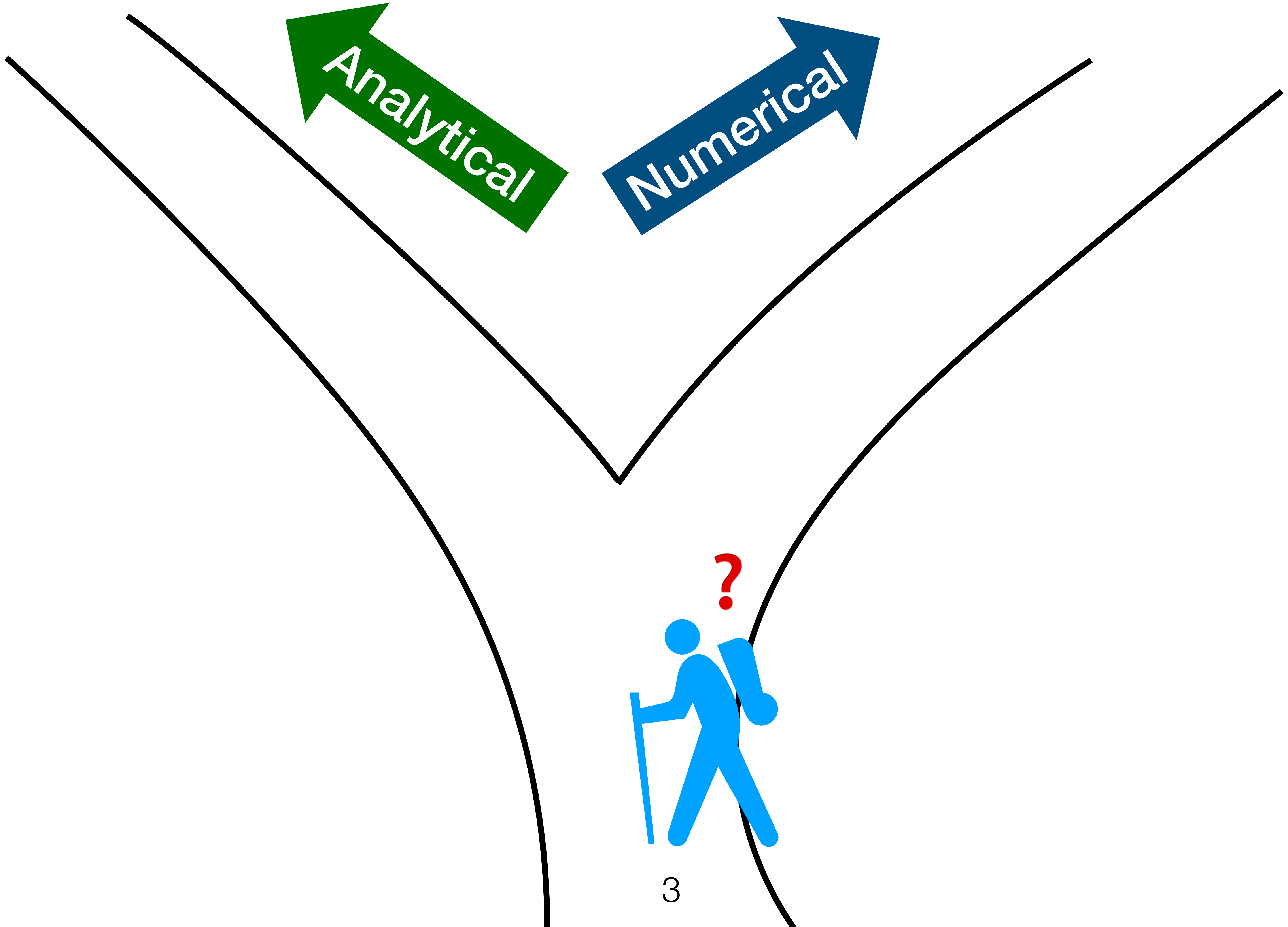
This is why we have "loops" in Loops and Legs!

Many techniques, yet they remain a bottleneck

One of the most powerful methods: integrals = solutions to **differential equations**

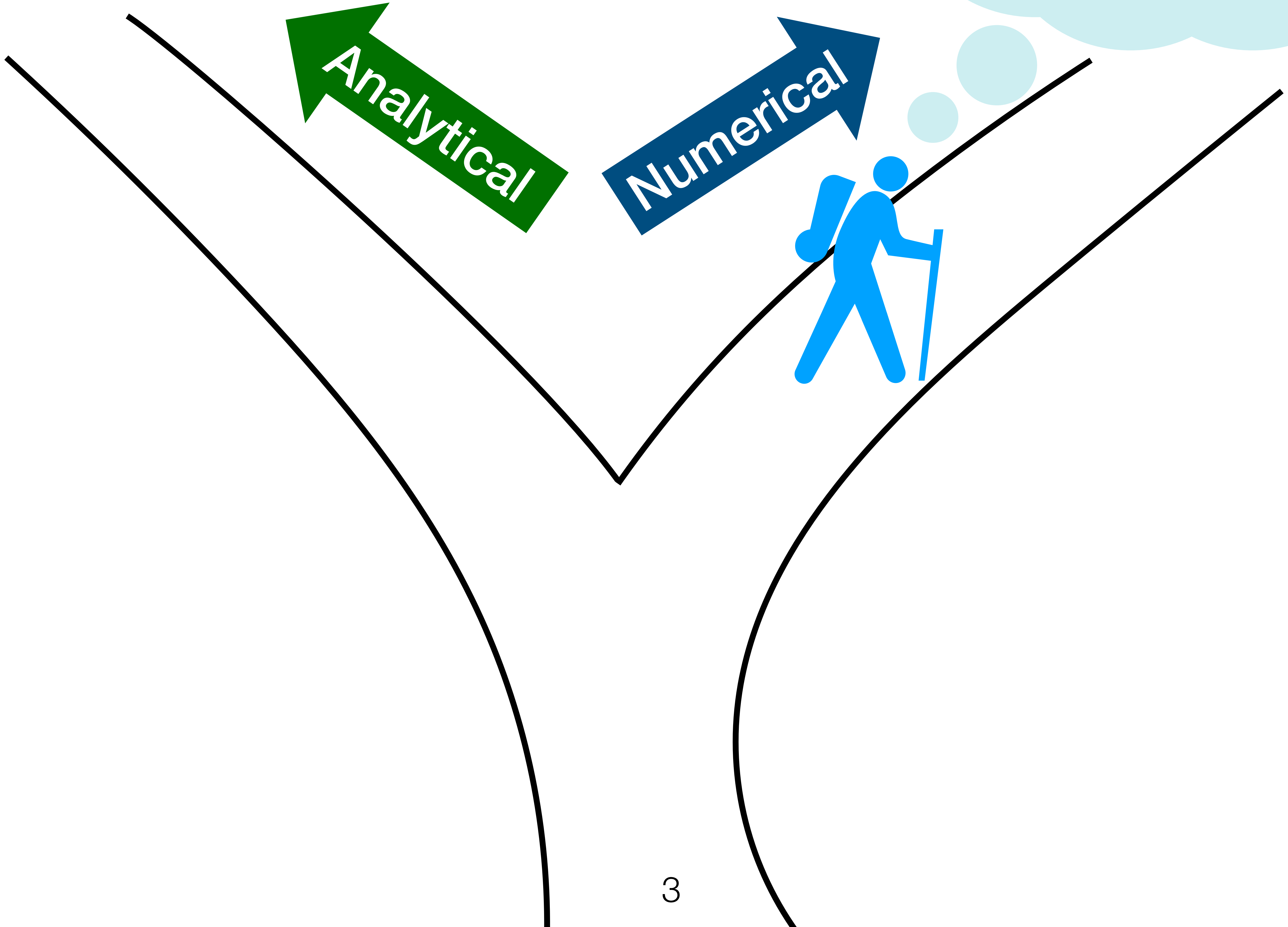
$$\frac{\partial}{\partial s}\overrightarrow{\mathrm{F}}(s;\epsilon) = A\,(s;\epsilon)\cdot\overrightarrow{\mathrm{F}}\,(s;\epsilon)$$

How do we solve the DEs?



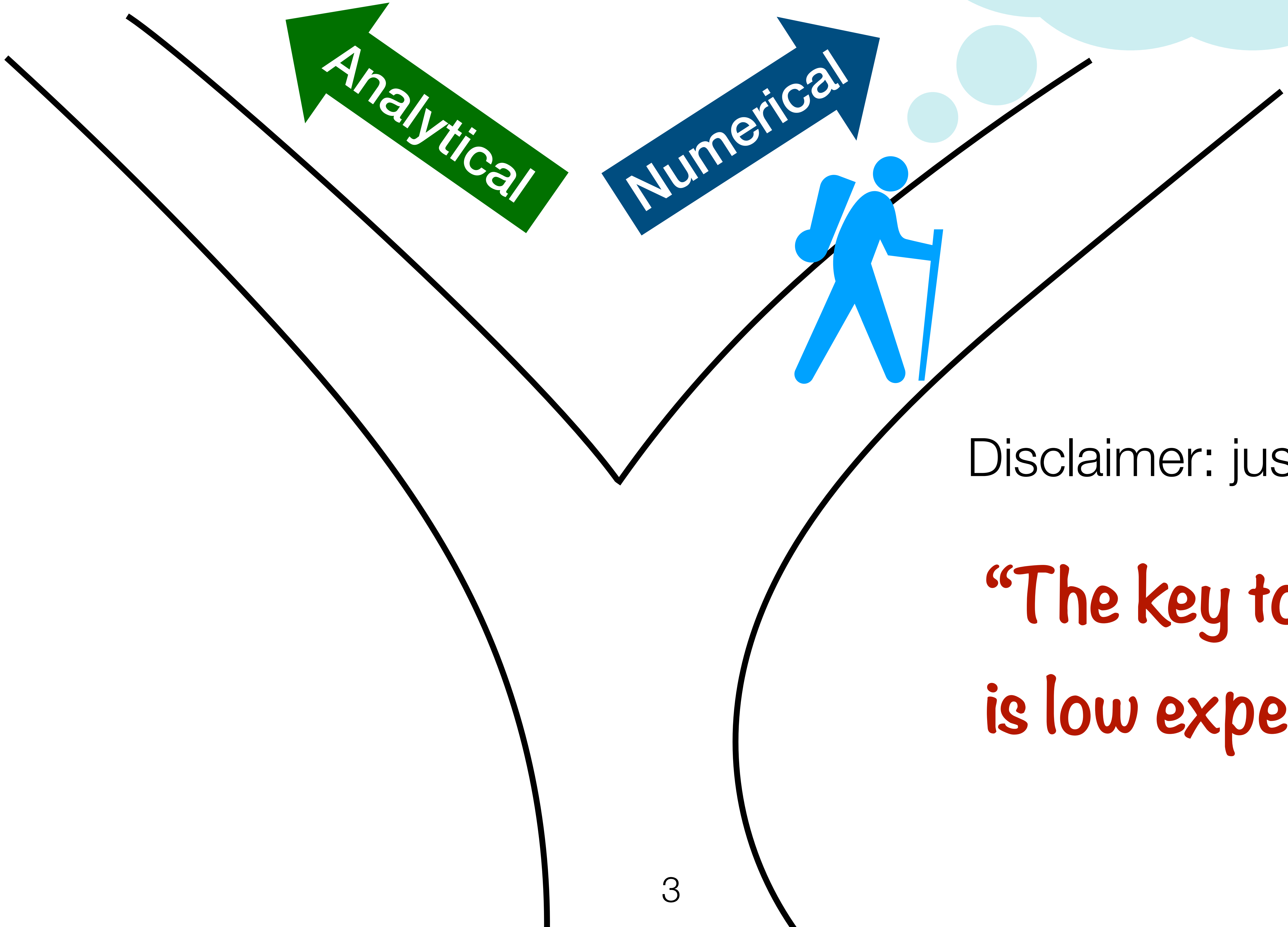
How do we solve th

Construct a **neural network** to approximate the solution



How do we solve th

Construct a **neural network** to approximate the solution



Disclaimer: just the first steps!

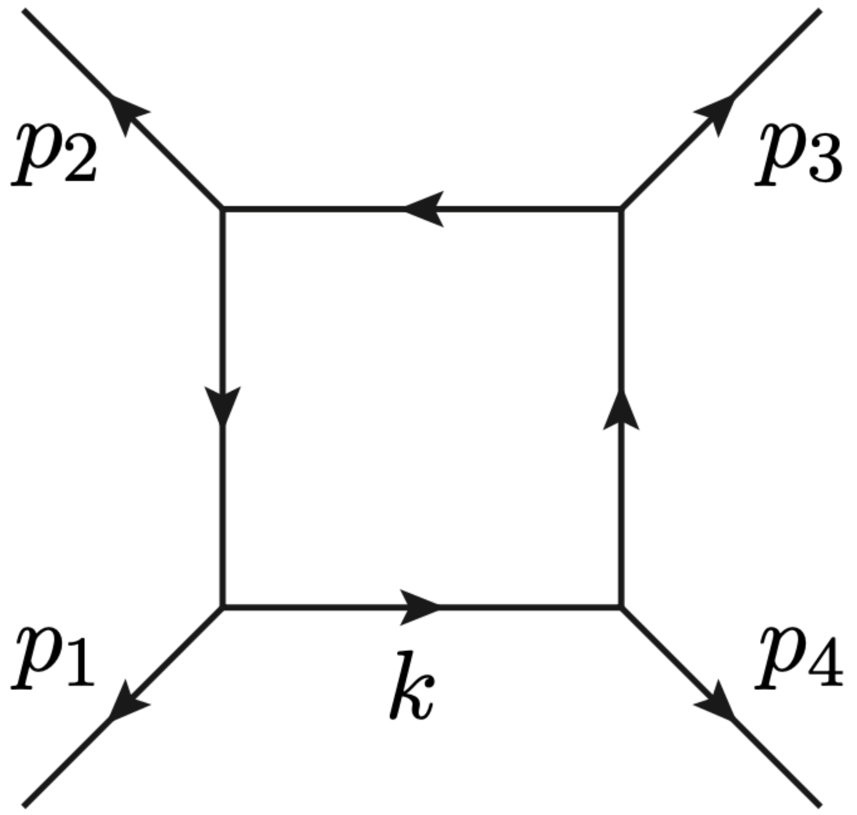
“The key to happiness is low expectations”

Method of differential equations

$$\frac{\partial}{\partial s_{12}} \overrightarrow{\mathrm{F}}(s;\epsilon) = A_{s_{12}}(s;\epsilon) \cdot \overrightarrow{\mathrm{F}}(s;\epsilon)$$

Integral families and master integrals

Scalar Feynman integrals with the same propagator structure = **integral family**



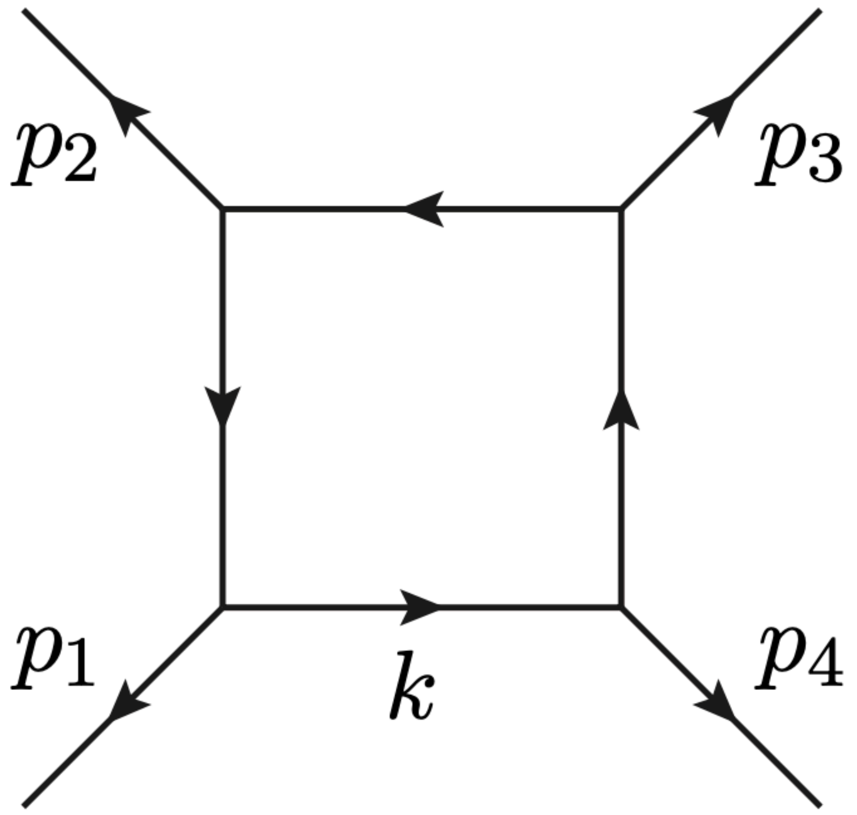
$$\mathrm{I}_{\vec{a}}(s,t;\epsilon) = \int \frac{\mathrm{d}^D k}{\mathrm{i}\pi^{D/2}} \frac{1}{D_1^{a_1} \ldots D_4^{a_4}}$$

$$\left\{ \mathrm{I}_{\vec{a}}(s,t;\epsilon) \,|\, \forall\, \vec{a} \in \mathbb{Z}^4 \right\}$$

$$D_1 = k^2$$
$$D_2 = (k+p_1)^2$$
$$D_3 = (k+p_1+p_2)^2$$
$$D_4 = (k-p_4)^2$$

Integral families and master integrals

Scalar Feynman integrals with the same propagator structure = **integral family**

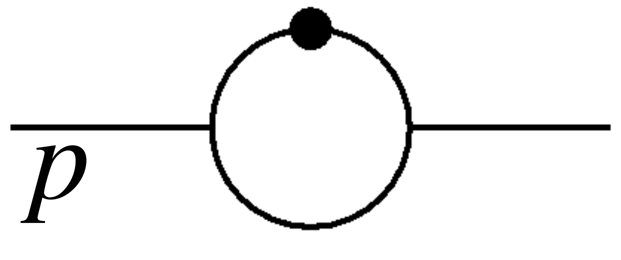


$$\mathrm{I}_{\vec{a}}(s,t;\epsilon) = \int \frac{\mathrm{d}^D k}{\mathrm{i}\pi^{D/2}} \frac{1}{D_1^{a_1}\ldots D_4^{a_4}}$$

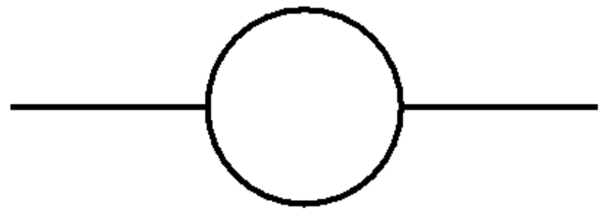
$$\left\{\mathrm{I}_{\vec{a}}(s,t;\epsilon)\,|\,\forall\,\vec{a}\in\mathbb{Z}^4\right\}$$

$$D_1 = k^2$$
$$D_2 = (k+p_1)^2$$
$$D_3 = (k+p_1+p_2)^2$$
$$D_4 = (k-p_4)^2$$

Identities among the $\mathrm{I}_{\vec{a}}$'s



$$= \frac{3-D}{p^2} \times$$

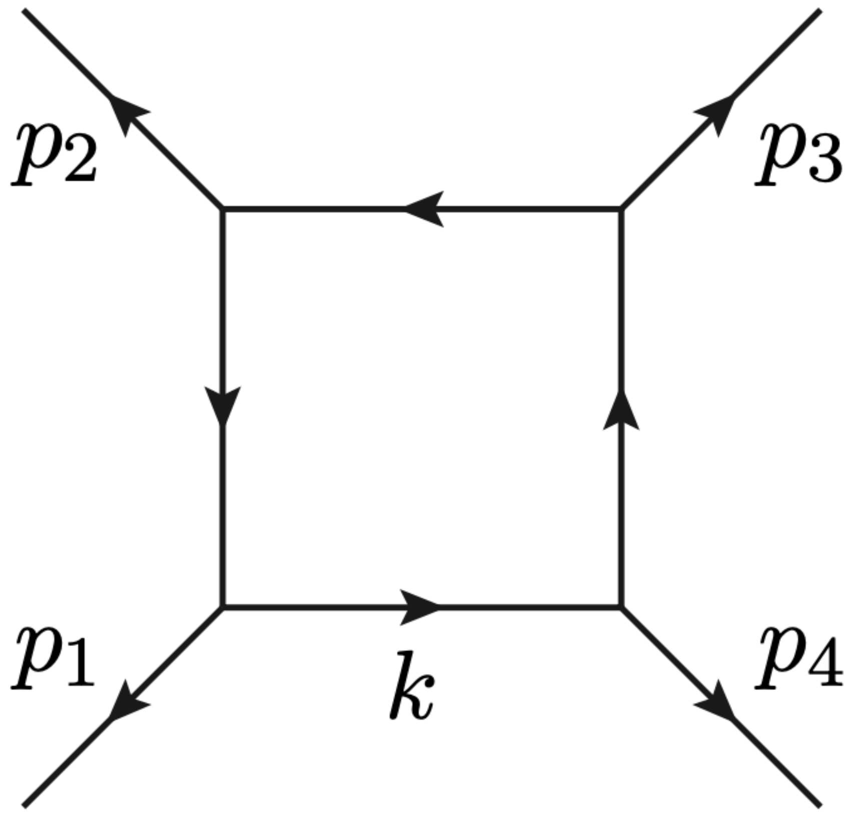


e.g. Integration-By-Parts relations

Chetyrkin, Tkachov '81; Laporta 2000

Integral families and master integrals

Scalar Feynman integrals with the same propagator structure = **integral family**

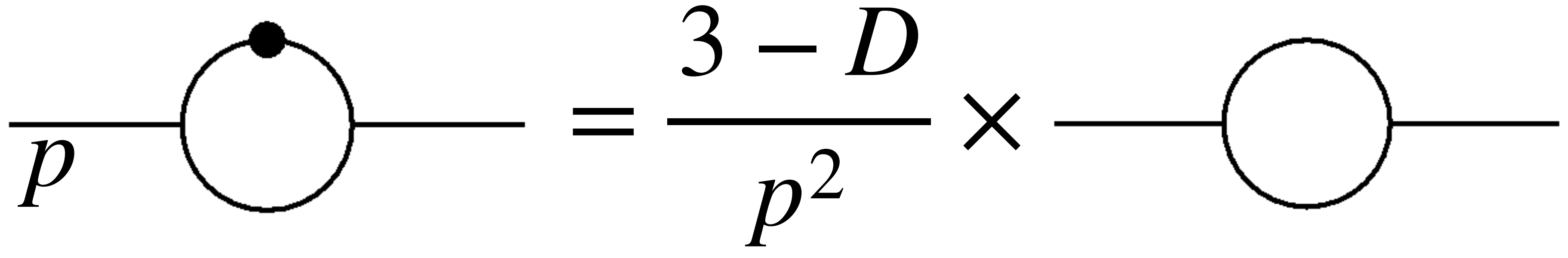


$$\mathrm{I}_{\vec{a}}(s,t;\epsilon) = \int \frac{\mathrm{d}^D k}{\mathrm{i}\pi^{D/2}} \frac{1}{D_1^{a_1} \ldots D_4^{a_4}}$$

$$D_1 = k^2$$
$$D_2 = (k+p_1)^2$$
$$D_3 = (k+p_1+p_2)^2$$
$$D_4 = (k-p_4)^2$$

$$\left\{ \mathrm{I}_{\vec{a}}(s,t;\epsilon) \,|\, \forall\, \vec{a} \in \mathbb{Z}^4 \right\}$$

Identities among the $\mathrm{I}_{\vec{a}}$'s



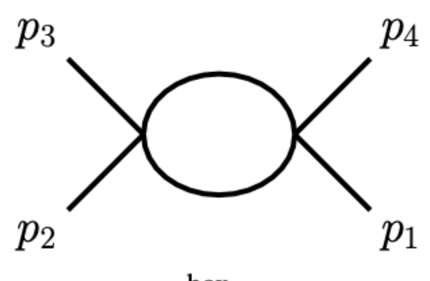
$$= \frac{3-D}{p^2} \times$$

e.g. Integration-By-Parts relations

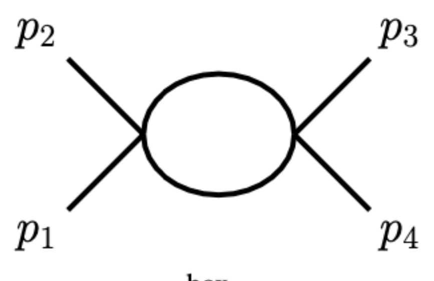
Chetyrkin, Tkachov '81; Laporta 2000

$\Rightarrow$

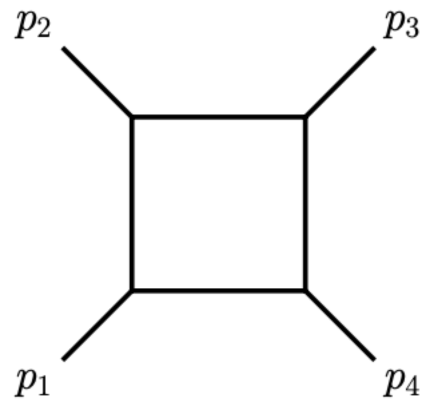
Finite-dimensional basis:
master integrals $\overrightarrow{\mathrm{F}}(s,t;\epsilon)$



$I^{\text{box}}_{0,1,0,1}$



$I^{\text{box}}_{1,0,1,0}$



$I^{\text{box}}_{1,1,1,1}$

Integrating by differentiating

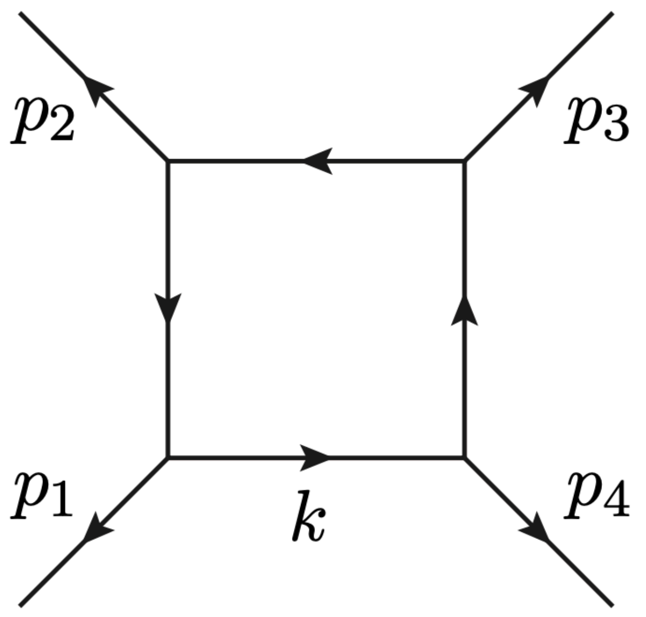
Barucchi, Ponzano '73; Kotikov '91; Bern, Dixon, Kosower '94; Gehrmann, Remiddi 2000

$$\frac{\partial}{\partial s_{12}} \overrightarrow{\mathrm{F}}(s;\epsilon) = \sum_{\vec{a}} c_{\vec{a}}\, \mathrm{I}_{\vec{a}}$$

IBP reduction

$$= A_{s_{12}}(s;\epsilon) \cdot \overrightarrow{\mathrm{F}}\,(s;\epsilon)$$

$$\frac{\partial}{\partial s}\vec{F}(s,t;\epsilon) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -\frac{\epsilon}{s} & 0 \\ \frac{2(2\epsilon-1)}{st(s+t)} & \frac{2(1-2\epsilon)}{s^2(s+t)} & -\frac{s+t+\epsilon\, t}{s(s+t)} \end{pmatrix} \cdot \vec{F}(s,t;\epsilon)$$



⇒ System of 1st order linear PDEs for the MIs $\overrightarrow{\mathrm{F}}$

How do we solve it? $\overrightarrow{\mathrm{F}}(s;\epsilon) = \sum_{w \geq w_{\min}} \epsilon^{w}\, \overrightarrow{\mathrm{F}}^{(w)}(s)$

Analytic solution not always feasible

Choose MIs such that DEs take **canonical form** *Henn 2013* No general algorithm!

Solution in terms of **special functions**

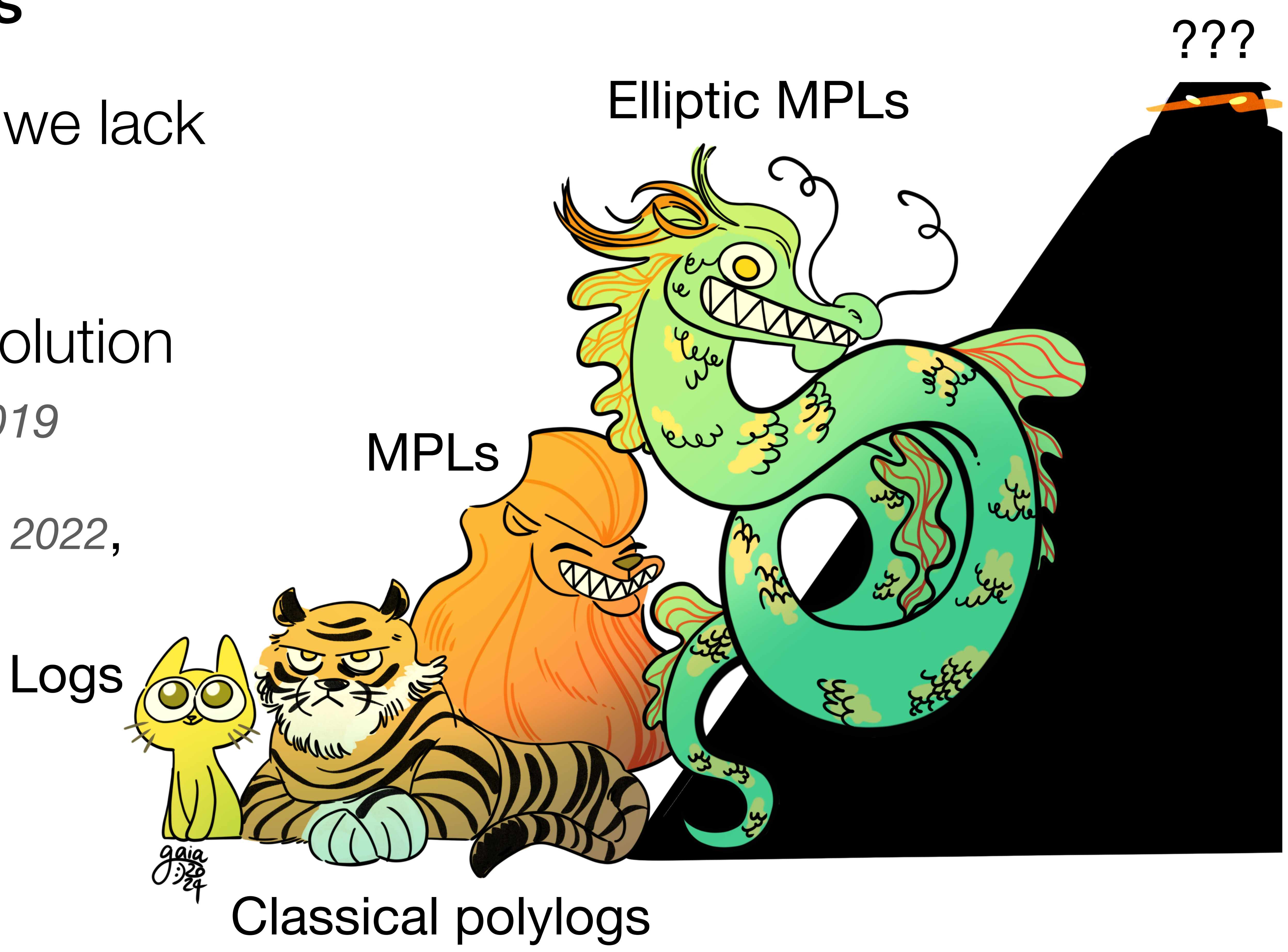
In the most complicated cases, we lack the mathematical technology!

Growing interest for semi-numerical solution based on series expansions *Moriello 2019*

DiffExp *Hidding 2020*, SeaSyde *Armadillo et al. 2022*, AMFlow *Ma, Liu 2022*

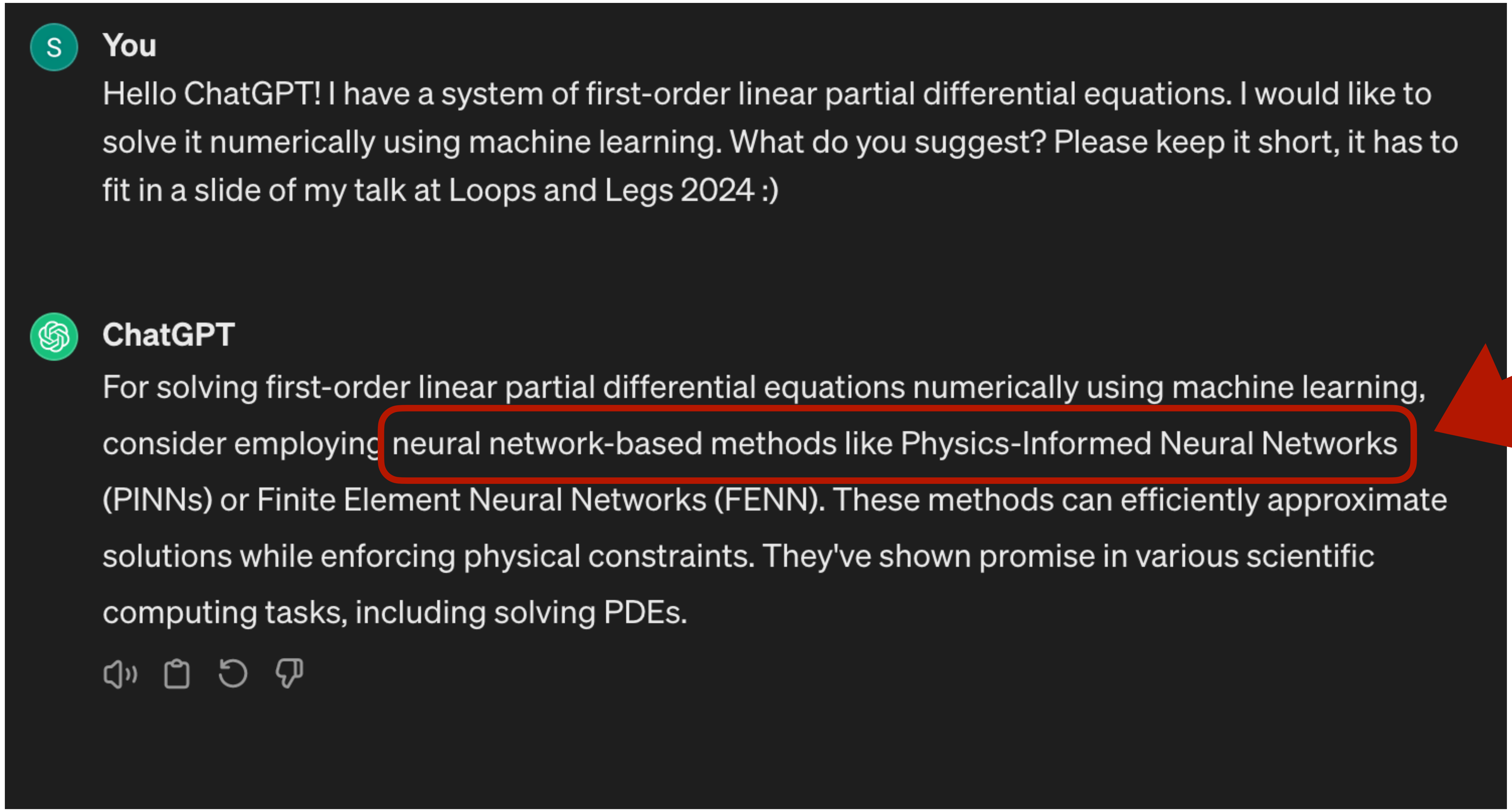
😄 Very flexible

🙁 Long evaluation times



Goals: flexibility + fast evaluation time

Can machine learning help to achieve this? Let's ask ChatGPT



You

Hello ChatGPT! I have a system of first-order linear partial differential equations. I would like to solve it numerically using machine learning. What do you suggest? Please keep it short, it has to fit in a slide of my talk at Loops and Legs 2024 :)

ChatGPT

For solving first-order linear partial differential equations numerically using machine learning, consider employing neural network-based methods like Physics-Informed Neural Networks (PINNs) or Finite Element Neural Networks (FENN). These methods can efficiently approximate solutions while enforcing physical constraints. They've shown promise in various scientific computing tasks, including solving PDEs.

Just what we ended up using!

We should have asked ChatGPT rightaway…

Slide idea from Melissa van Beekveld

Neural networks are universal function approximators

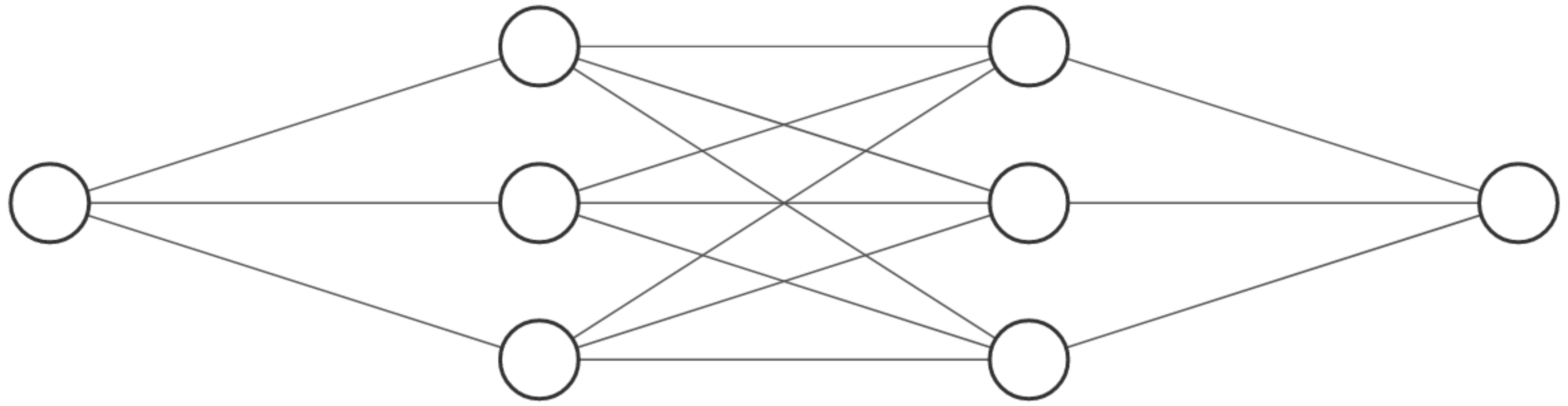
Hornik, Stinchcombe, White ’89

Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$

Neural networks are universal function approximators

Hornik, Stinchcombe, White '89

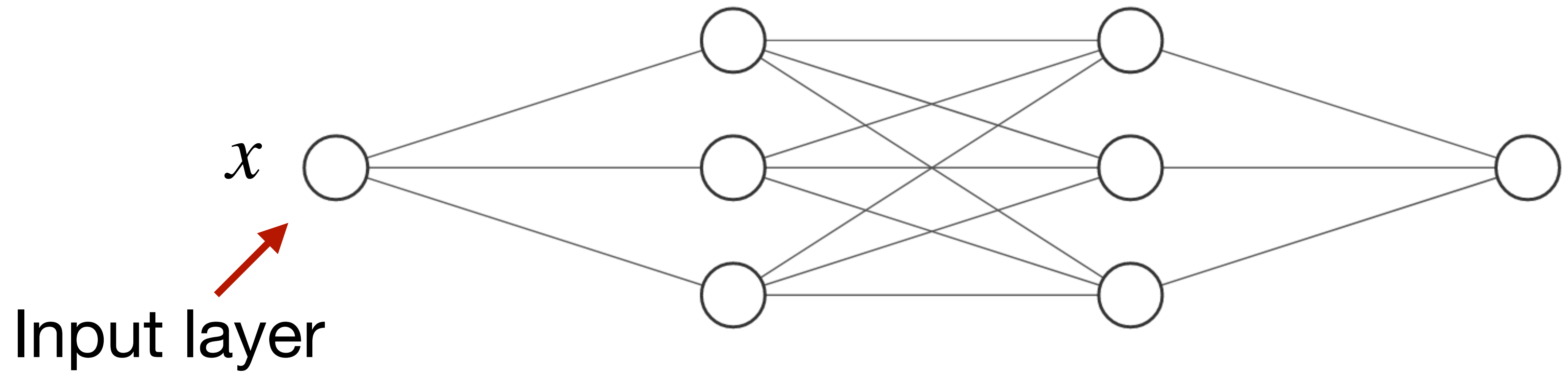
Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Neural networks are universal function approximators

Hornik, Stinchcombe, White ’89

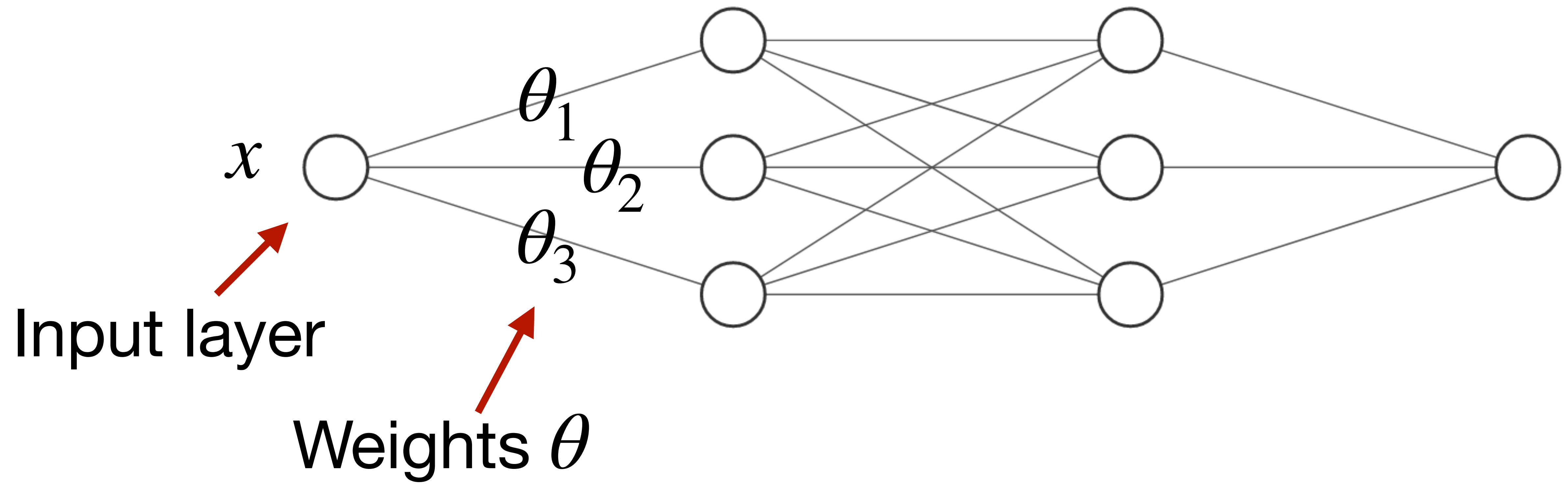
Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Neural networks are universal function approximators

Hornik, Stinchcombe, White ’89

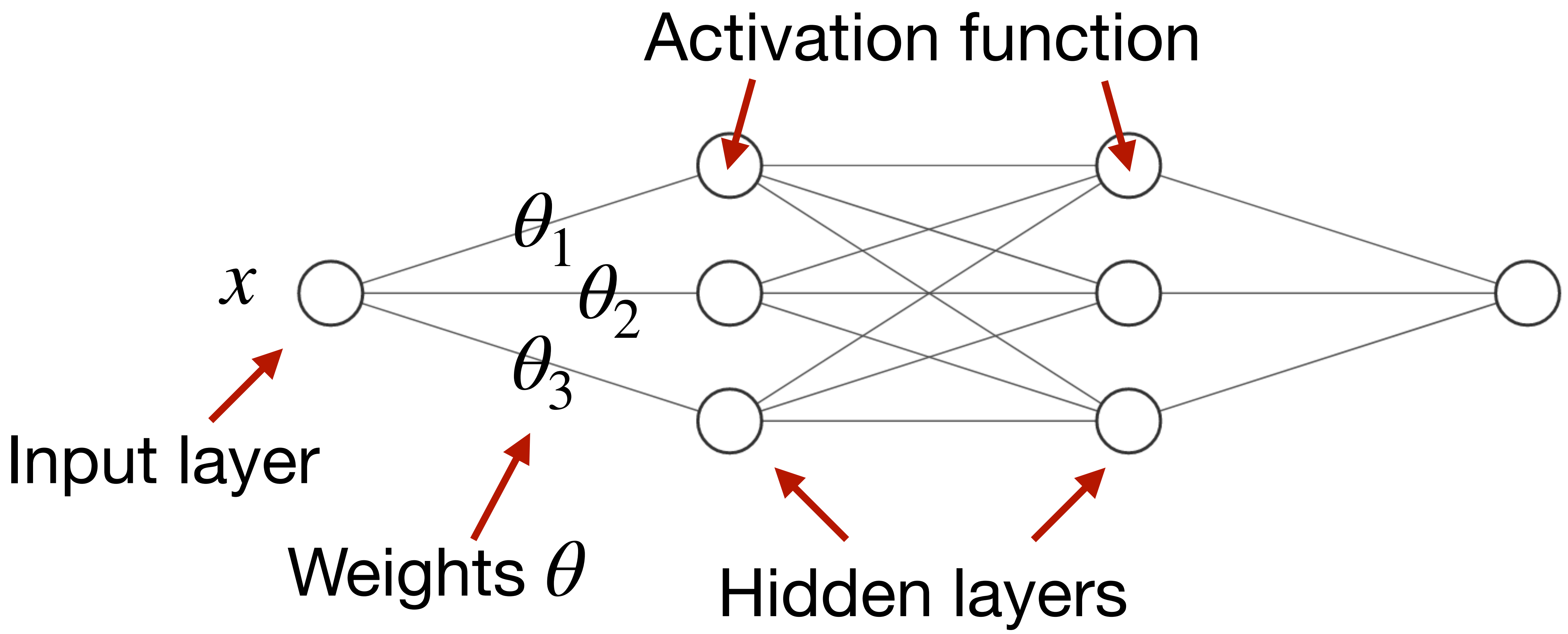
Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Neural networks are universal function approximators

Hornik, Stinchcombe, White ’89

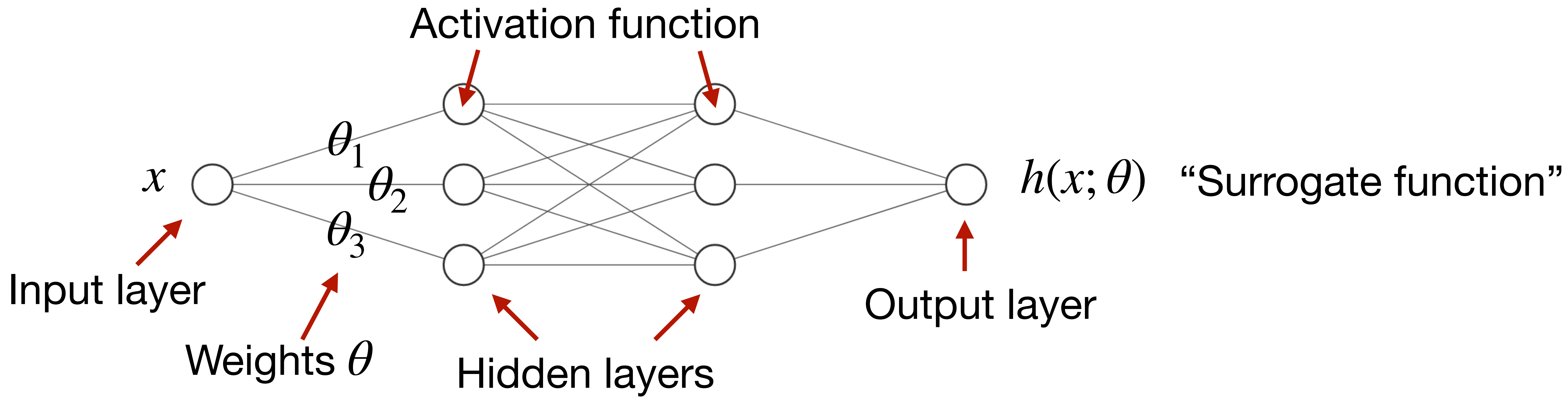
Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Neural networks are universal function approximators

Hornik, Stinchcombe, White ’89

Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Activation function

x

θ_1

θ_2

θ_3

$h(x;\theta)$ “Surrogate function”

Input layer

Weights θ

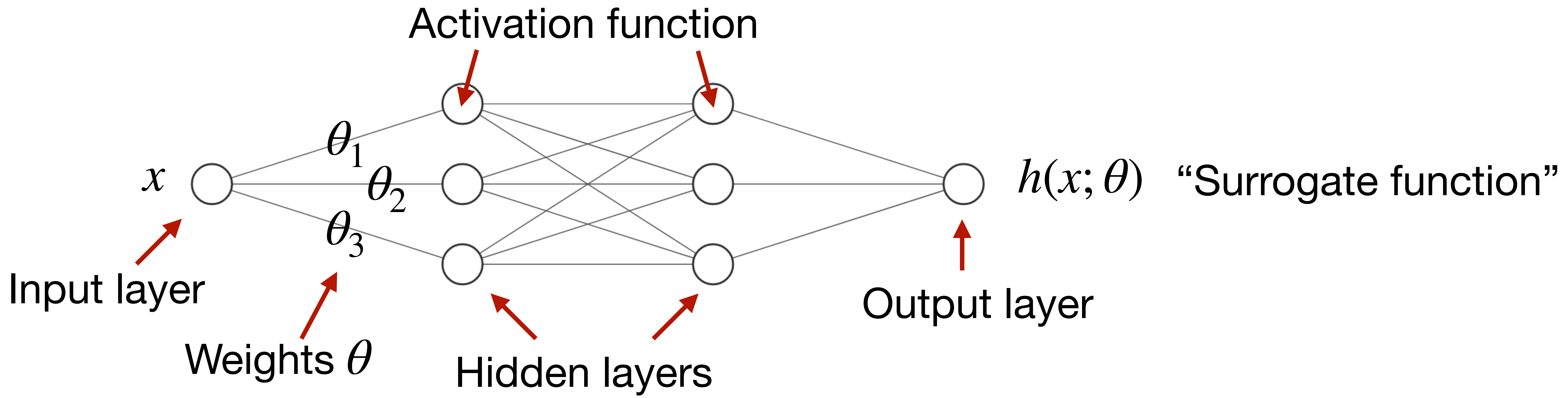
Hidden layers

Output layer

Neural networks are universal function approximators

Hornik, Stinchcombe, White '89

Typical problem: approximate function $f(x)$ from large dataset of values $f(x_i)$



Optimisation problem: find weights θ such that a **loss function** is minimised

$$L(\mathrm{D};\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[f(x_i) - h(x_i;\theta)\right]^2$$

We don’t have a large dataset…

What we have:

- Small dataset of values (at least 1), obtained numerically in other ways

  E.g. AMFlow *Liu, Ma 2022* → Expensive evaluation, but very flexible

- Differential equations: $$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = A(x)\,f(x)$$

Physics-informed deep learning

Raissi, Perdikaris, Karniadakis 2017

💡 Idea: include the DEs in the loss function

$$L(\mathrm{D};\theta) = \overline{\sum_i} \left[h(x_i;\theta) - f(x_i)\right]^2 + \overline{\sum_j}\left[\left.\frac{\mathrm{d}h(x;\theta)}{\mathrm{d}x}\right|_{x=x_j} - A(x_j)\,h(x_j;\theta)\right]^2$$

Small “boundary” dataset

Infinite dimensional “DE” dataset

Derivatives of the NN computed with automatic differentiation *Griewank, Walther 2008*

Input: few boundary values + the analytic DEs

The canonical form of the DEs is not needed

We make mild assumptions to simplify the problem:

$$\frac{\partial}{\partial v_i}\overrightarrow{F}(\vec{v};\epsilon) = A_{v_i}(\vec{v};\epsilon)\cdot\overrightarrow{F}(\vec{v};\epsilon) \quad \forall\, i = 1,\ldots,n_v$$

$\vec{v}$: kinematic variables

The canonical form of the DEs is not needed

We make mild assumptions to simplify the problem:

$$\frac{\partial}{\partial v_i}\vec{F}(\vec{v};\epsilon) = A_{v_i}(\vec{v};\epsilon)\cdot\vec{F}(\vec{v};\epsilon) \quad \forall\, i = 1,\ldots,n_v$$

$\vec{v}$: kinematic variables

1. The matrices $A_{v_i}(\vec{v};\epsilon)$ are rational functions ⇒ Separate Re/Im parts, only deal with real numbers

$$\frac{\partial}{\partial v_i}\mathrm{Re}\left[\vec{F}(\vec{v};\epsilon)\right] = A_{v_i}(\vec{v};\epsilon)\cdot\mathrm{Re}\left[\vec{F}(\vec{v};\epsilon)\right]$$

$$\frac{\partial}{\partial v_i}\mathrm{Im}\left[\vec{F}(\vec{v};\epsilon)\right] = A_{v_i}(\vec{v};\epsilon)\cdot\mathrm{Im}\left[\vec{F}(\vec{v};\epsilon)\right]$$

The canonical form of the DEs is not needed

We make mild assumptions to simplify the problem:

$$\frac{\partial}{\partial v_i} \overrightarrow{F}(\vec{v};\epsilon) = A_{v_i}(\vec{v};\epsilon)\cdot \overrightarrow{F}(\vec{v};\epsilon) \quad \forall\, i = 1,\ldots,n_v$$

$\vec{v}$: kinematic variables

1. The matrices $A_{v_i}(\vec{v};\epsilon)$ are rational functions ⇒ Separate Re/Im parts, only deal with real numbers

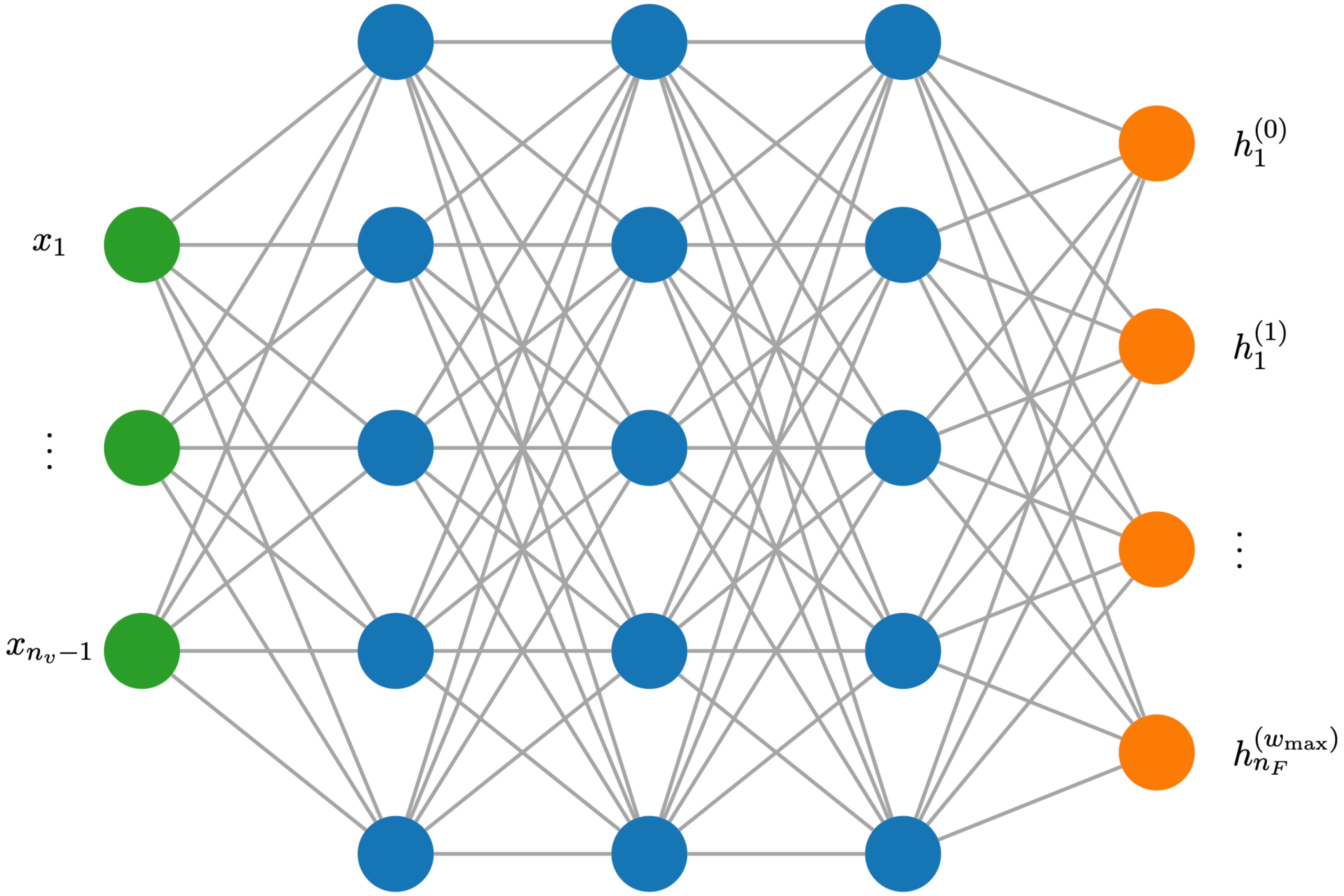
2. The matrices $A_{v_i}(\vec{v};\epsilon)$ are finite at $\epsilon = 0$, $A_{v_i}(\vec{v};\epsilon) = \sum_{k=0}^{k_{\max}} \epsilon^k\, A_{v_i}^{(k)}(\vec{v})$

⇒ Simplifies the ϵ expansion of the solution $\overrightarrow{F}(\vec{v};\epsilon) = \epsilon^{w*} \sum_{w=0}^{w_{\max}} \epsilon^w\, \overrightarrow{F}^{(w)}(\vec{v})$

Architecture

PyTorch

Dimensionless
kinematic
variables



Re or Im part of
$\vec{F}^{(w)}$ up to a
certain order in ϵ

In the examples we considered: 3/4 hidden layers, 32—256 nodes per layer

Our loss function in full glory

$$L_{\mathrm{b}}(\mathrm{D}_{\mathrm{b}},\theta) = \mathop{\overline{\sum}}_{\vec{x}^{(i)}\in\mathrm{D}_{\mathrm{b}}} \mathop{\overline{\sum}}_{j=1}^{n_F}\mathop{\overline{\sum}}_{w=0}^{w_{\max}} \left[h_j^{(w)}\left(\vec{x}^{(i)};\theta\right) - g_j^{(w)}\left(\vec{x}^{(i)}\right)\right]^2$$

Either Re of Im part of the master integrals

Fixed small database of known values

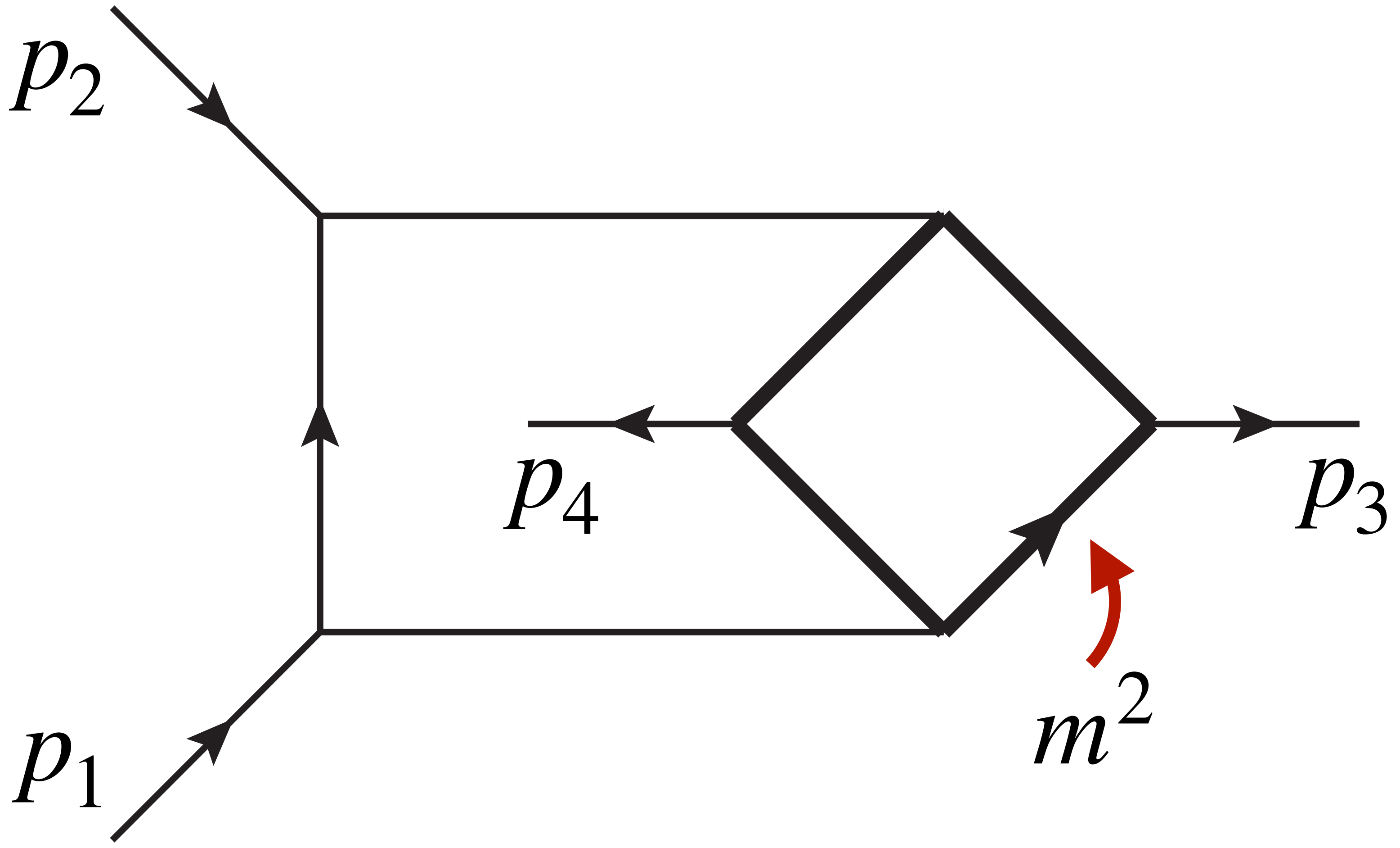
$$L_{\mathrm{DE}}(\mathrm{D}_{\mathrm{DE}},\theta) =$$

$$\mathop{\overline{\sum}}_{\vec{x}^{(i)}\in\mathrm{D}_{\mathrm{DE}}} \mathop{\overline{\sum}}_{j=1}^{n_F}\mathop{\overline{\sum}}_{l=1}^{n_v-1}\mathop{\overline{\sum}}_{w=0}^{w_{\max}} \left[\partial_{x_l} h_j^{(w)}\left(\vec{x}^{(i)};\theta\right) - \sum_{k=0}^{\min(w,k_{\max})}\sum_{r=1}^{n_F} A^{(k)}_{x_l,jr}\left(\vec{x}^{(i)}\right) h_r^{(w-k)}\left(\vec{x}^{(i)};\theta\right)\right]^2$$

Dynamic random sampling at each iteration

- Avoids over-fitting, no regularisation needed
- Validation can be done on the training dataset

Heavy crossed box



3 kinematic variables, 36 MIs

$$\vec{v} = \left\{s = (p_1 + p_2)^2 \,,\ t = (p_1 - p_3)^2 \,,\ m^2\right\}$$

Canonical DEs / analytic solution unavailable

Involves elliptic functions

von Manteuffel, Tancredi 2017; Xu, Yang 2019; Wang, Wang, Xu, Xu, Yang 2021; Görges, Nega, Tancredi, Wagner 2023; Ahmed, Chaubey, Kaur, Maggio 2024

→ Ekta Chaubey's talk

Full computation recently using generalised power series expansions (DiffExp)

Becchetti, Bonciani, Cieri, Coro, Ripani 2023

Hidding 2020

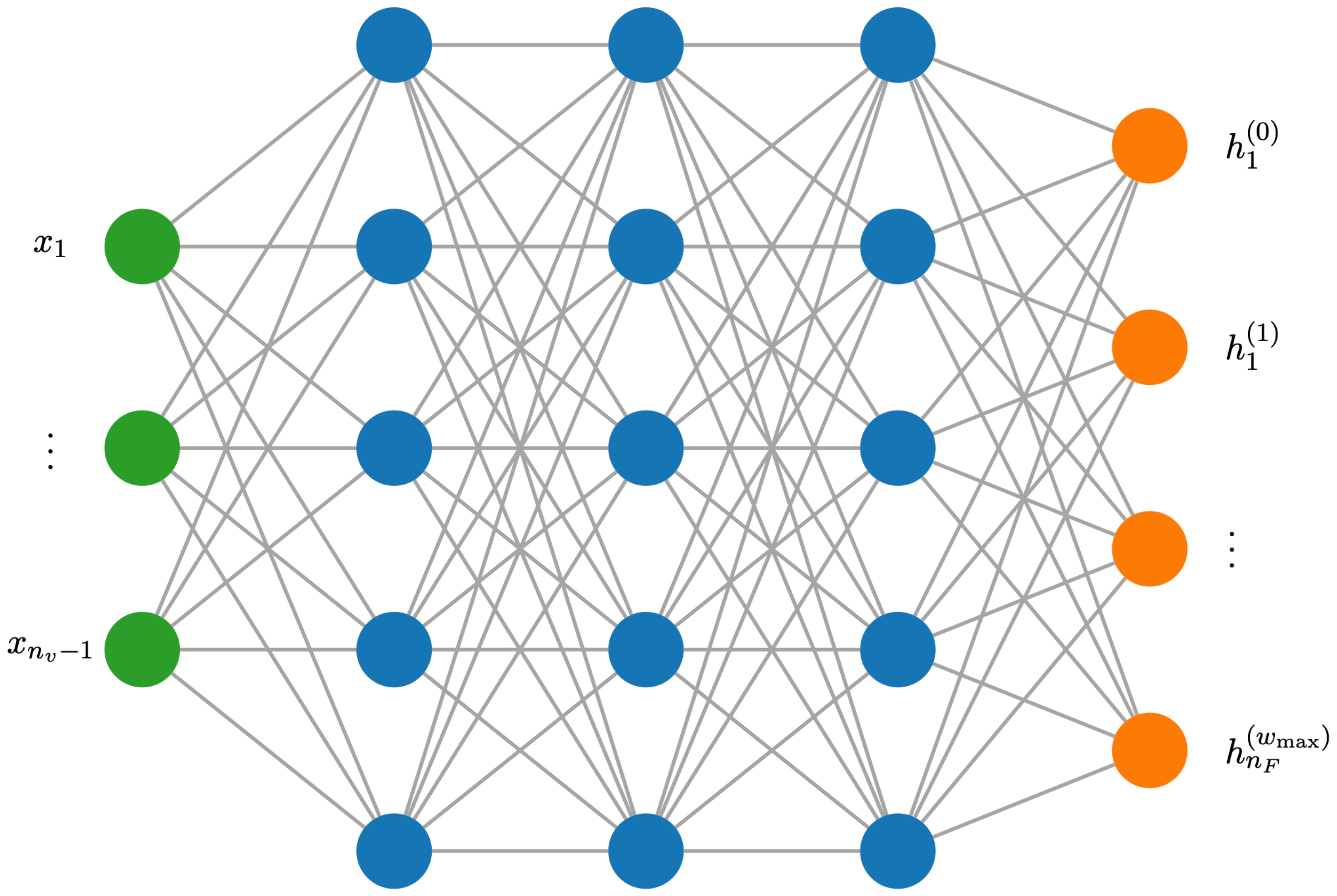
MIs stripped of square roots →

$$A_{v_i}(\vec{v};\epsilon) = \sum_{k=0}^{2} \epsilon^k A^{(k)}_{v_i}(\vec{v})$$

→ Federico Coro's talk

Heavy crossed box: architecture

2 input variables
(fix $m^2 = 1$)



3 hidden layers, 256 neurons each

MIs (Re or Im)

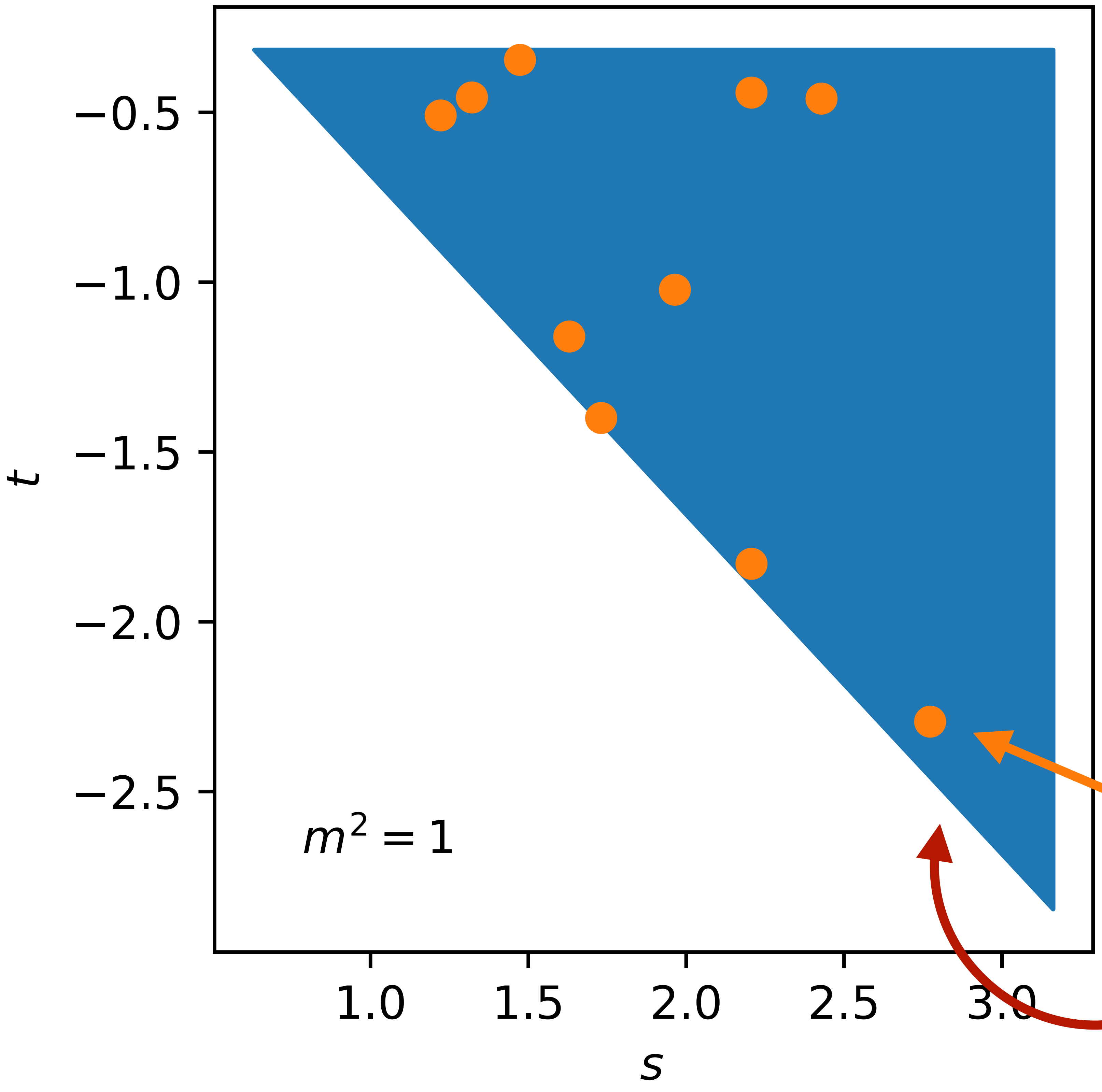
36 x 5 = 180 outputs

ϵ orders

$$\vec{F}(\vec{v};\epsilon) = \frac{1}{\epsilon^4}\sum_{w=0}^{4} \epsilon^w \, \vec{F}^{(w)}(\vec{v})$$

Heavy crossed box: kinematic region

s channel: $s > -t > 0 \wedge m^2 > 0$ → Never leave the chosen domain of analyticity domain, so analytic continuation is not required



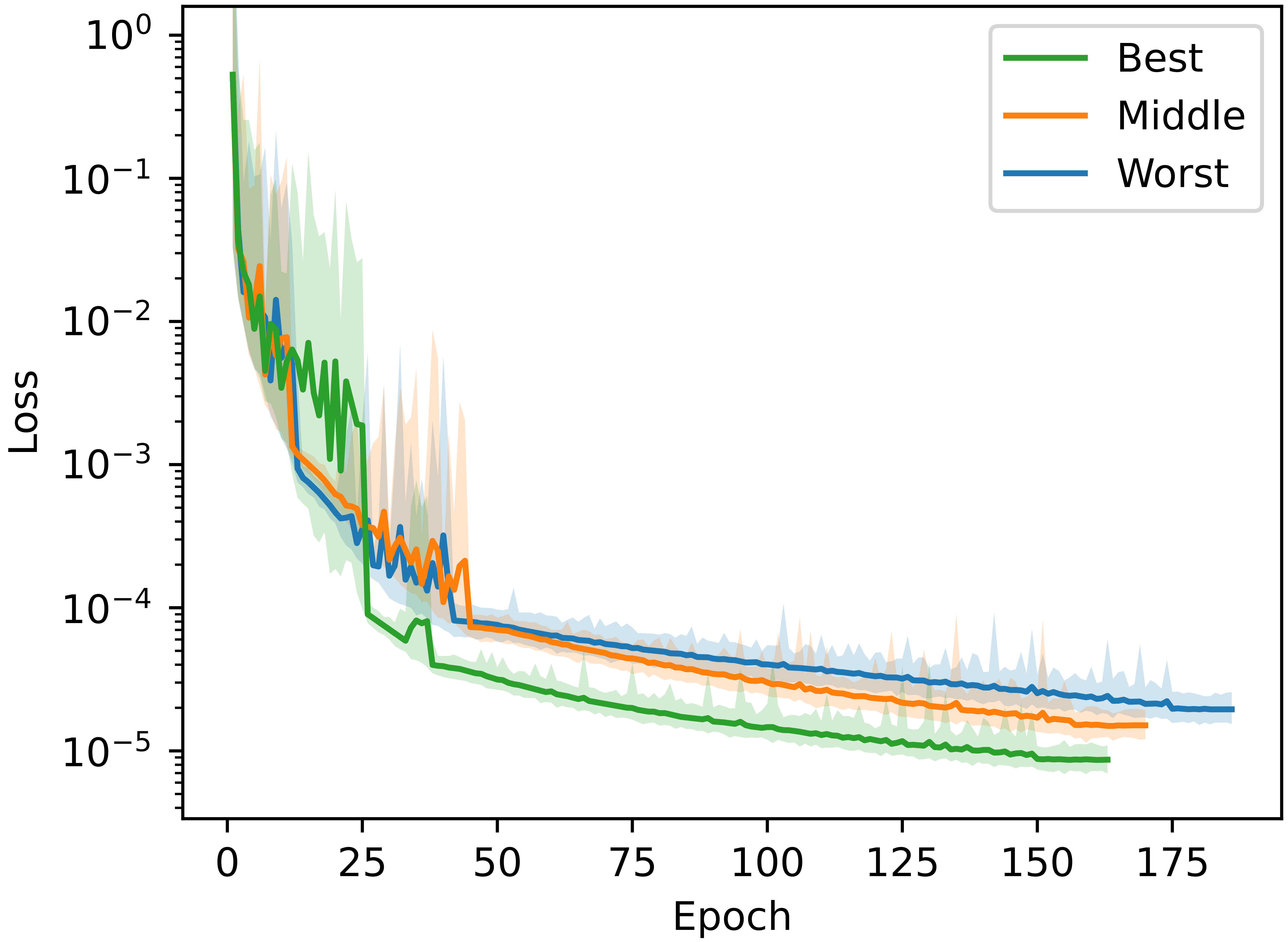
We choose $s < \sqrt{10}$

Singularities of the solution

Cut near boundaries:
$10\,\%$ of largest value ($\sqrt{10}$)

Boundary values at 10 random points, obtained with AMFlow *Liu, Ma 2022*

Heavy crossed box: training



Ensemble of 10 NNs

Iterations: 7.9×10^4

Time to train 1 NN: 75 min
(on a good laptop, GPU)

Use training metric for validation, as inputs for DE loss function are dynamically random sampled

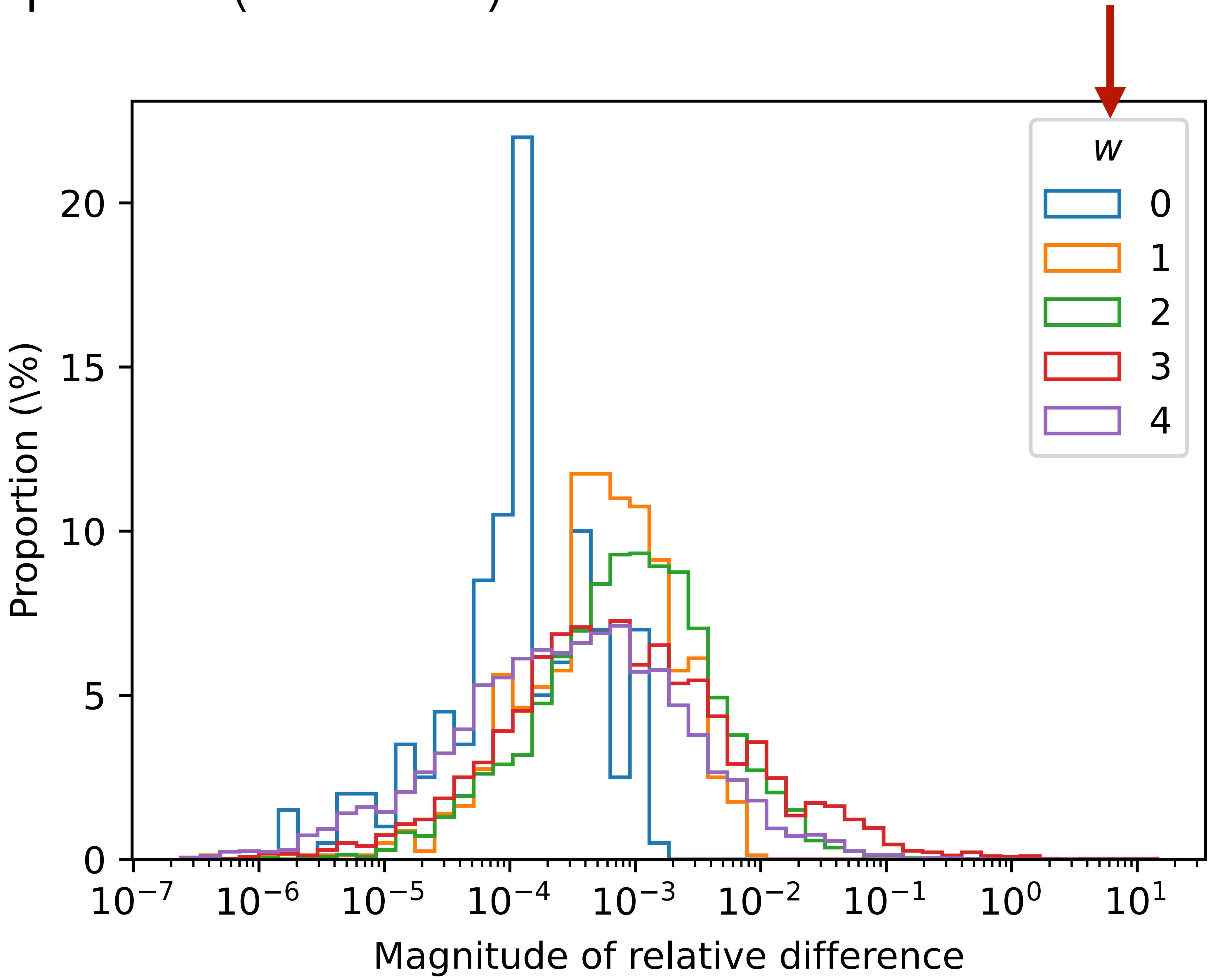
Heavy crossed box: model performance

Comparison against testing dataset of 100 points (AMFlow)

Mean absolute difference: 1.6×10^{-3}

Mean magnitude of rel. diff.: 7.3×10^{-3}

Evaluation time $\sim 1 - 10\ \mu s$



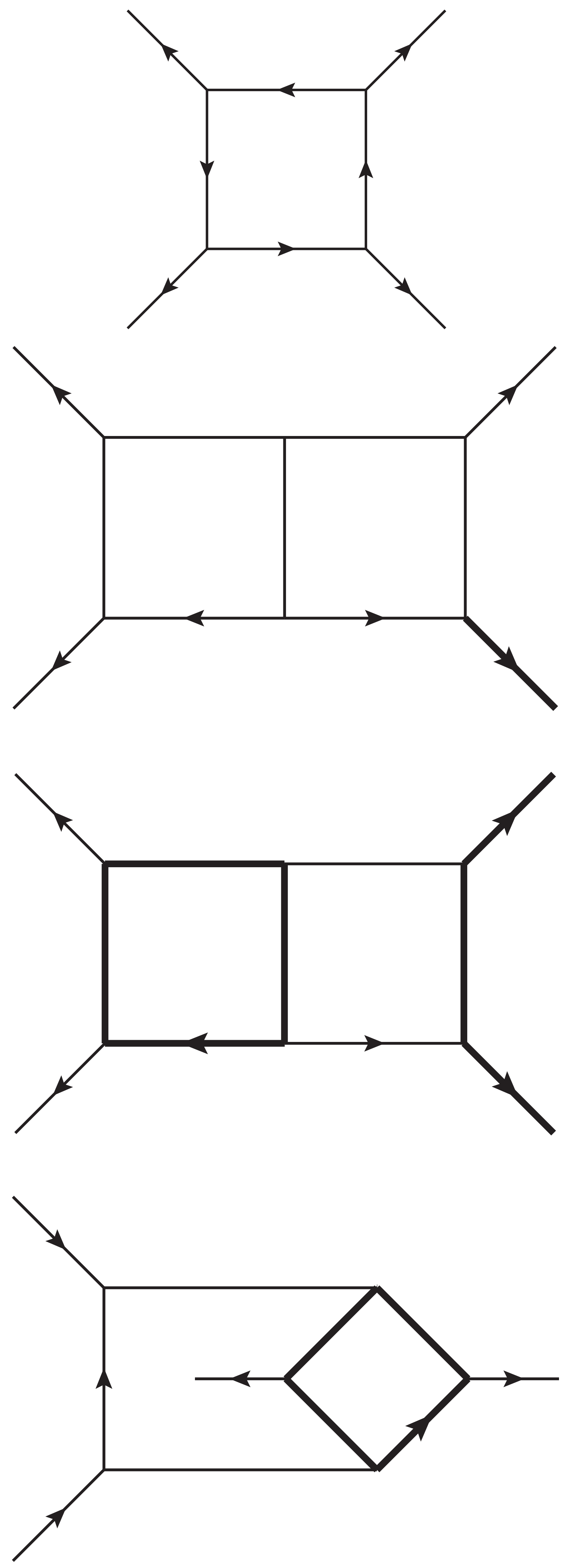
ϵ orders

Good and bad

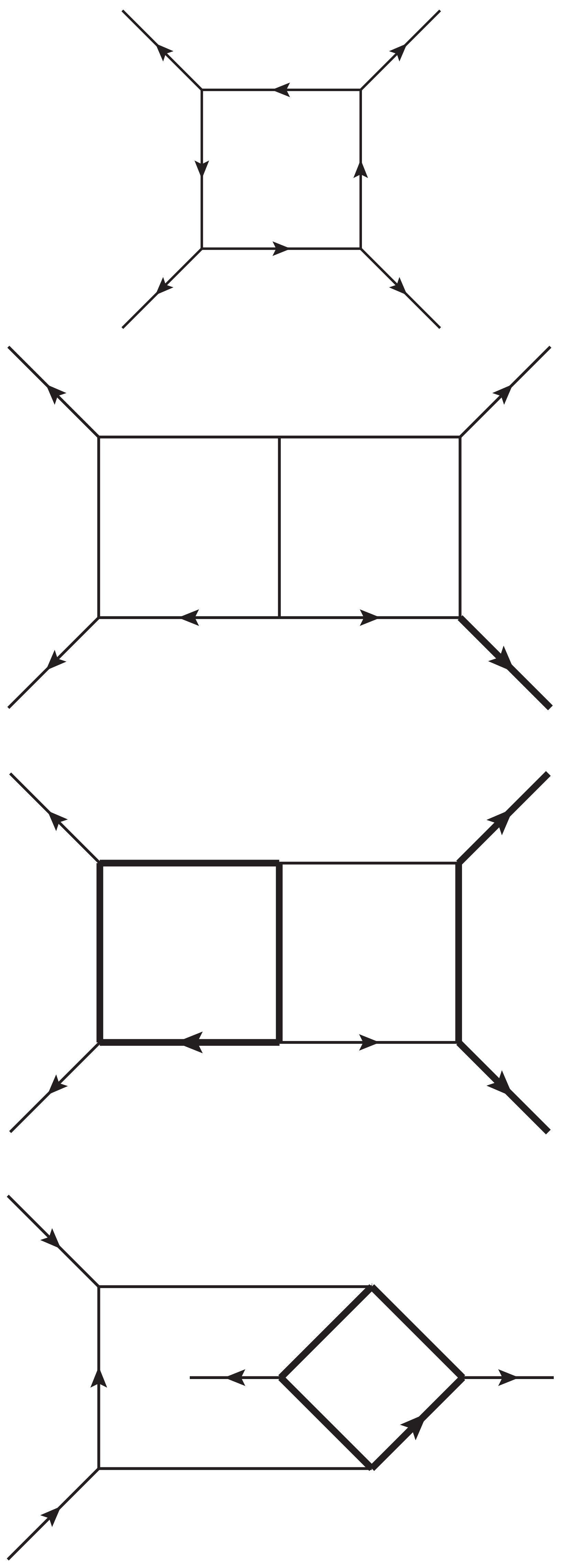
Flatness of the performance with respect to

- Analytic complexity (ϵ orders, MI) within the same family
- Across different families

Instantaneous evaluation times 🥳



Good and bad



Flatness of the performance with respect to

- Analytic complexity (ϵ orders, MI) within the same family
- Across different families

Instantaneous evaluation times 🥳

As of now, low control over accuracy 🙁

We can estimate it (ensemble uncertainty, differential error), but as of now unclear how to increase it arbitrarily

Good and bad

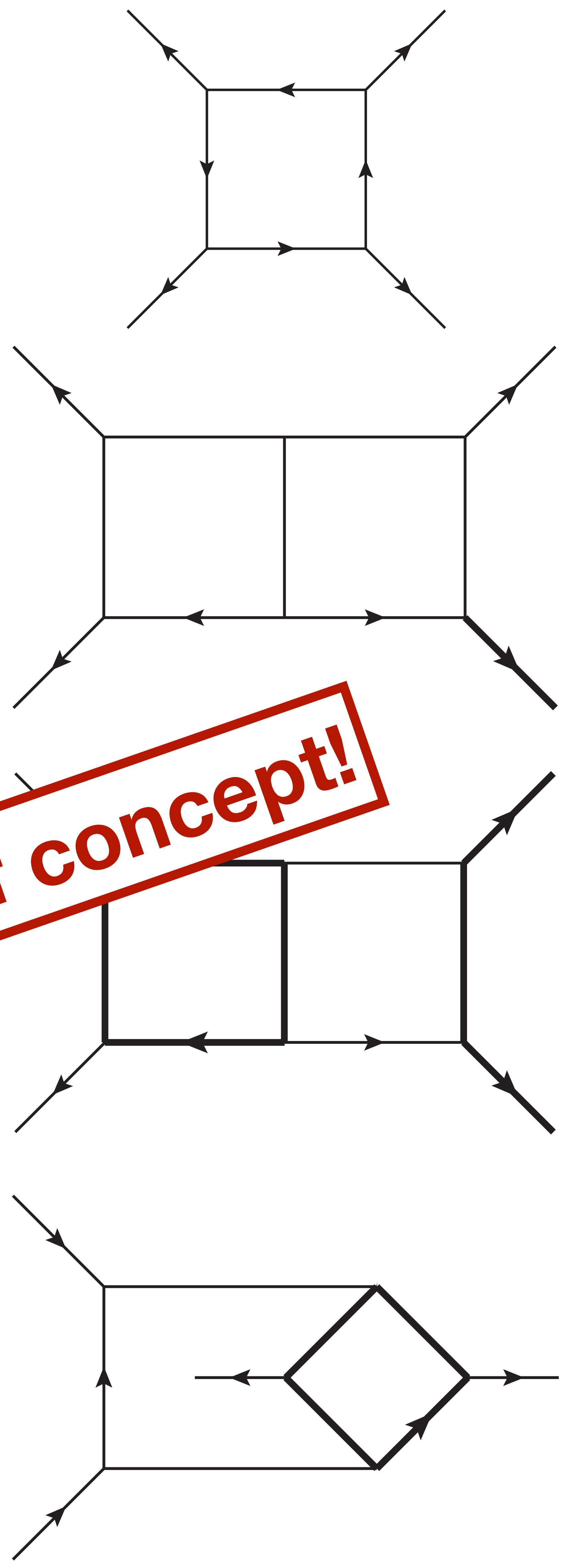
Flatness of the performance with respect to

- Analytic complexity (ϵ orders, MI) within the same family
- Across different families

Instantaneous evaluation times 🥳

As of now, low control over accuracy 🙁

We can estimate it (ensemble uncertainty, differential error), but as of now unclear how to increase it arbitrarily



Only proof of concept!

Good and bad

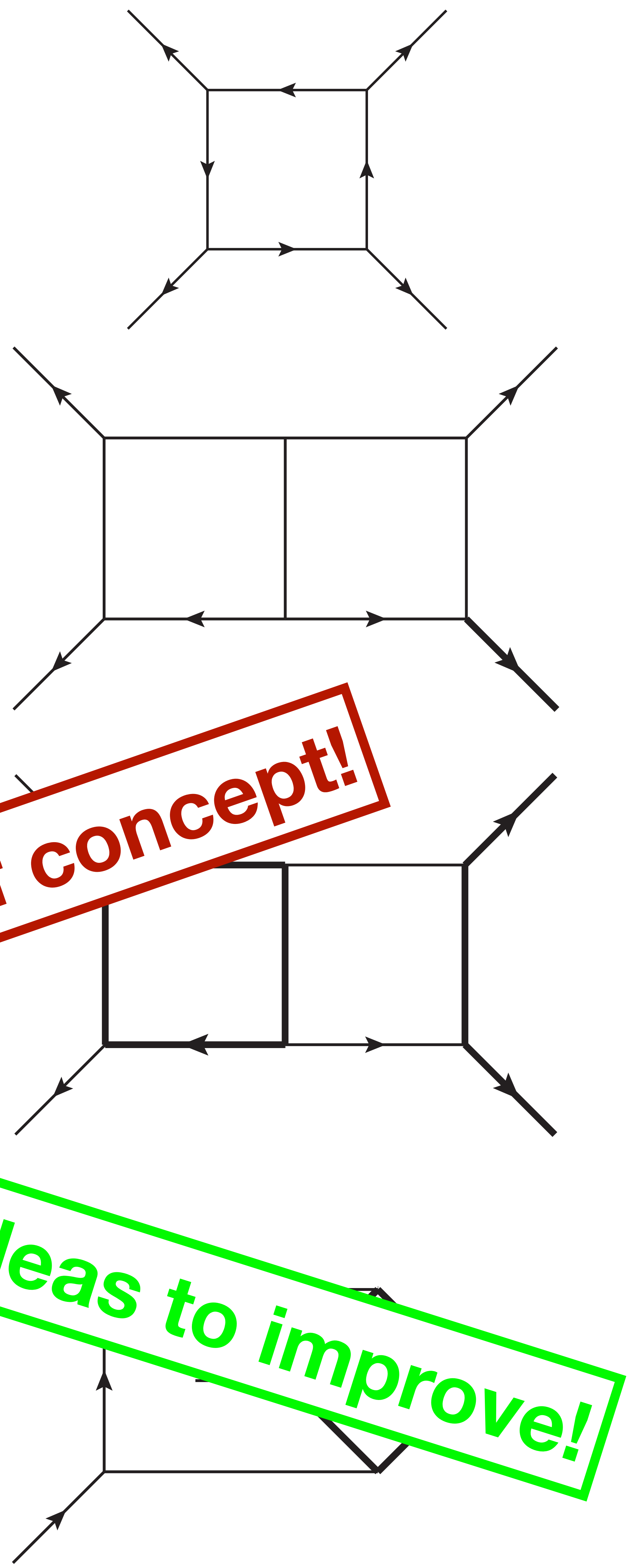
Flatness of the performance with respect to

- Analytic complexity (ϵ orders, MI) within the same family
- Across different families

Instantaneous evaluation times 🥳

As of now, low control over accuracy 🙁

We can estimate it (ensemble uncertainty, differential error), but as of now unclear how to increase it arbitrarily



Only proof of concept!

Many ideas to improve!

Conclusion

New method to evaluate numerically Feynman integrals satisfying generic DEs using physics-informed deep learning

Proof-of-concept implementation can reach 1% accuracy in non-trivial 2-loop examples

Much room for improvement!

Francesco Calisto, Ryan Moodie, **Simone Zoia**
(arXiv:2312.02067)

Conclusion

New method to evaluate numerically Feynman integrals satisfying generic DEs using physics-informed deep learning

Proof-of-concept implementation can reach 1% accuracy in non-trivial 2-loop examples

Much room for improvement!

Francesco Calisto, Ryan Moodie, **Simone Zoia**
(arXiv:2312.02067)

Thank you!

Proof-of-concept implementation

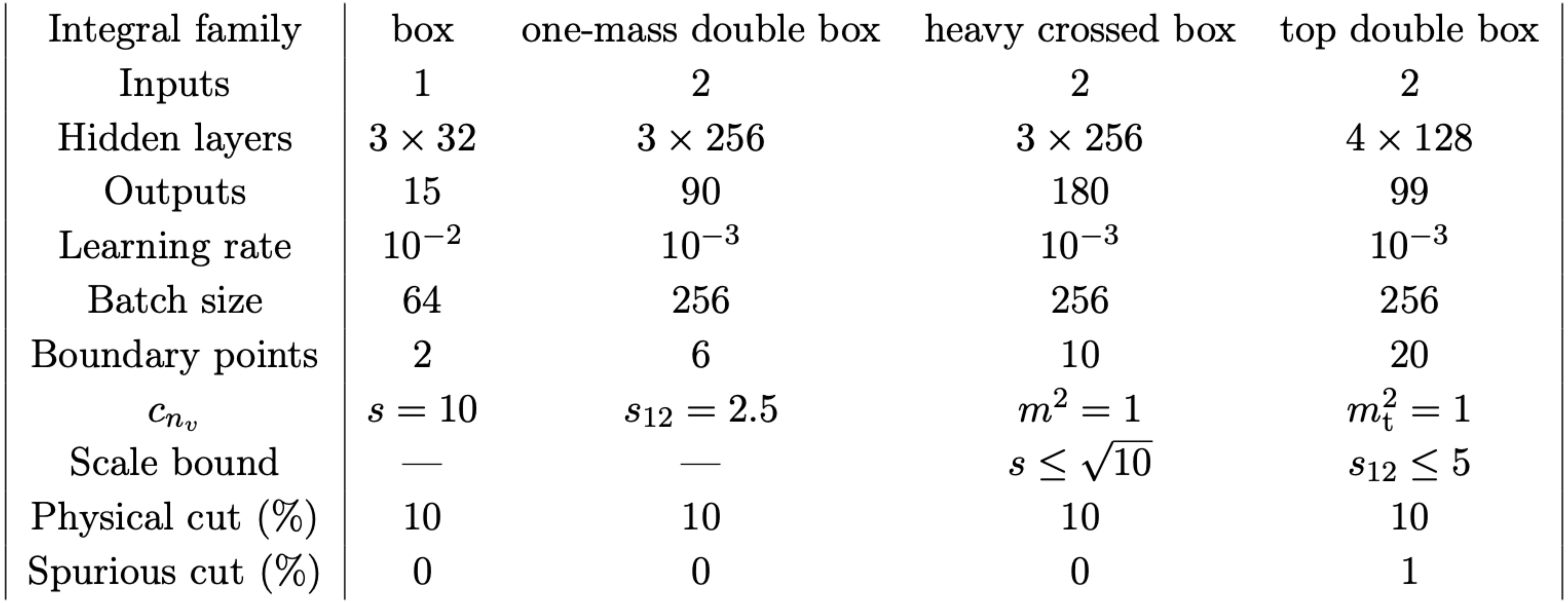
PyTorch

GELU (Gaussian Error Linear Unit) activation function (nonzero and continuous 2nd-order derivatives)

Train with stochastic gradient descent (Adam optimiser)

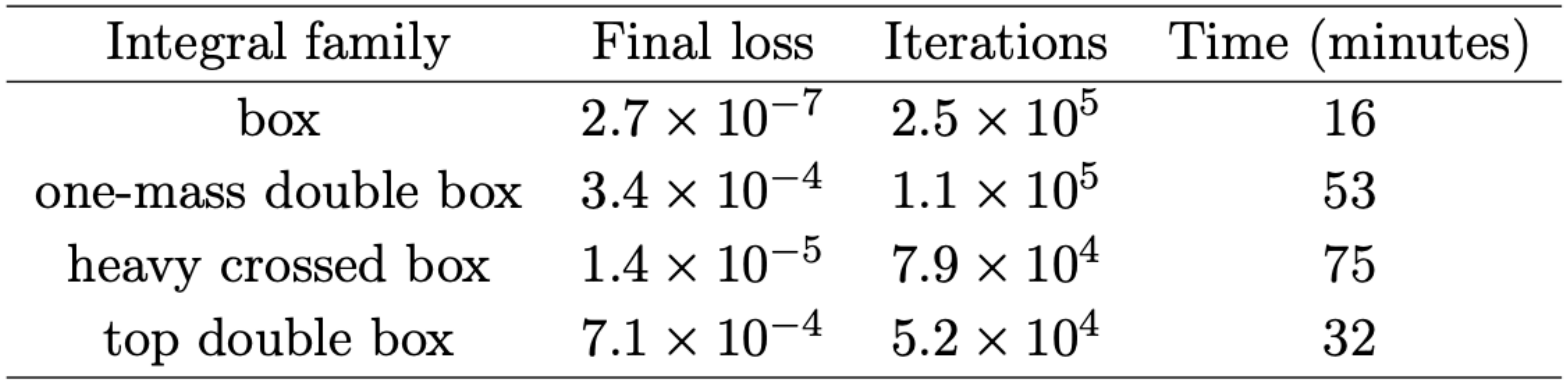
Mini-batch training: iterations organised into epochs composed of small batches, taking a dynamic random sample of the inputs for each batch

- No need for regularisation to avoid overfitting
- Validation can be done on the training dataset



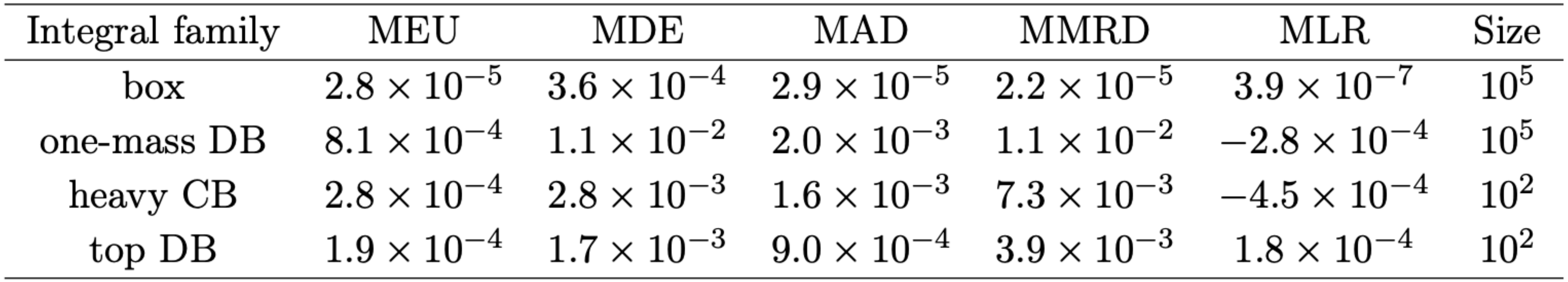
Integral family	box	one-mass double box	heavy crossed box	top double box
Inputs	1	2	2	2
Hidden layers	3×32	3×256	3×256	4×128
Outputs	15	90	180	99
Learning rate	10^{-2}	10^{-3}	10^{-3}	10^{-3}
Batch size	64	256	256	256
Boundary points	2	6	10	20
c_{n_v}	$s = 10$	$s_{12} = 2.5$	$m^2 = 1$	$m_{\mathrm{t}}^2 = 1$
Scale bound	—	—	$s \leq \sqrt{10}$	$s_{12} \leq 5$
Physical cut (%)	10	10	10	10
Spurious cut (%)	0	0	0	1

Summary of hyperparameters



Integral family	Final loss	Iterations	Time (minutes)
box	2.7×10^{-7}	2.5×10^{5}	16
one-mass double box	3.4×10^{-4}	1.1×10^{5}	53
heavy crossed box	1.4×10^{-5}	7.9×10^{4}	75
top double box	7.1×10^{-4}	5.2×10^{4}	32

Training statistics



Integral family	MEU	MDE	MAD	MMRD	MLR	Size
box	2.8×10^{-5}	3.6×10^{-4}	2.9×10^{-5}	2.2×10^{-5}	3.9×10^{-7}	10^{5}
one-mass DB	8.1×10^{-4}	1.1×10^{-2}	2.0×10^{-3}	1.1×10^{-2}	-2.8×10^{-4}	10^{5}
heavy CB	2.8×10^{-4}	2.8×10^{-3}	1.6×10^{-3}	7.3×10^{-3}	-4.5×10^{-4}	10^{2}
top DB	1.9×10^{-4}	1.7×10^{-3}	9.0×10^{-4}	3.9×10^{-3}	1.8×10^{-4}	10^{2}

Uncertainty and testing errors