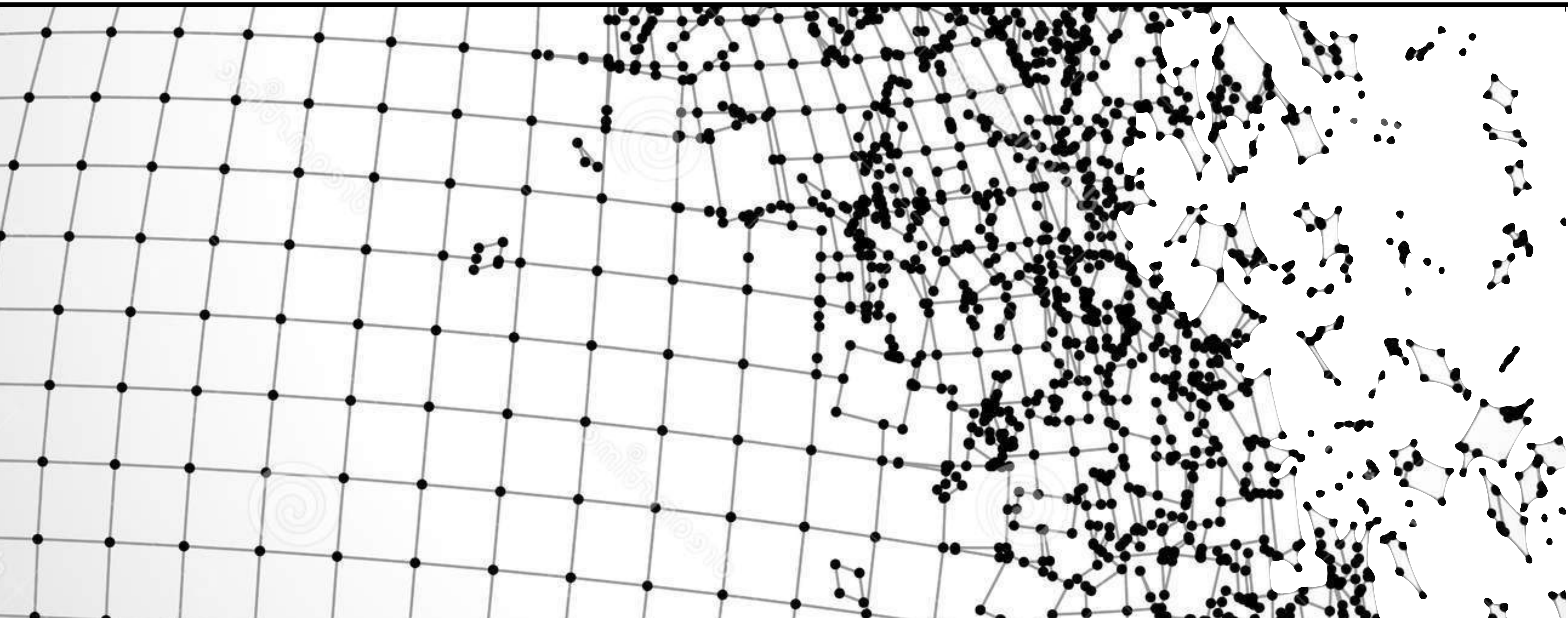
Breaking the Mesh:
Solving Partial Differential Equations with Deep Learning



James B. Scoggins and Loïc Gouarin

SMAI 2019 Mini-Symposium, Guidel Plages

17-19h, 13 May 2019

The Lineup

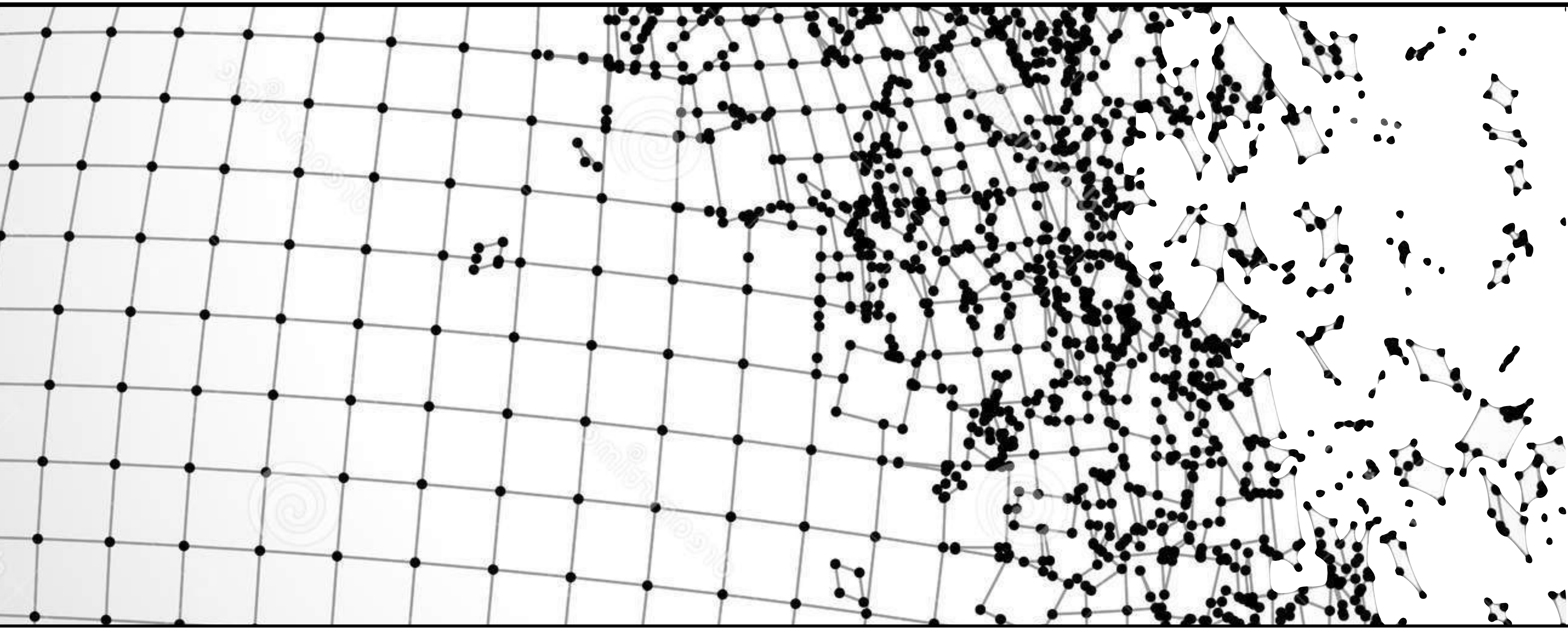
17:00 - **James B. Scoggins**
Postdoctoral Researcher at CMAP, Ecole Polytechnique, France
Solving partial differential equations with deep learning

17:30 - **Philippe Von Wurstemberger**
Doctoral Student at ETH Zurich, Switzerland
Overcoming the curse of dimensionality with DNNs: Theoretical approximation results for PDEs

18:00 - **Rémi Gribonval**
Research Director at INRIA in Rennes, France
Approximation spaces of deep neural networks

18:30 - **Siamak Mehrkanoon**
Assistant Professor at Maastricht University, The Netherlands
LS-SVM based solutions to differential equations

Solving Partial Differential Equations with Deep Learning



James B. Scoggins, Eric Moulines, Marc Massot

SMAI 2019, Guidel Plages

13 May 2019

École Polytechnique Université Paris-Saclay

CMAP

Partial differential equations permeate our world

They lay at the heart of predictive modeling

$$\frac{\partial \mathbf{u}}{\partial t} = \mathscr{F}[t, \mathbf{x}, \mathbf{u}, \nabla_{\mathbf{x}} \mathbf{u}, \ldots]$$

Partial differential equations permeate our world

They lay at the heart of predictive modeling

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{F}[t, \mathbf{x}, \mathbf{u}, \nabla_{\mathbf{x}} \mathbf{u}, \ldots]$$

Physical Law
The *rate of change* of a quantity over time is related to the local value of that quantity and how it changes in space.

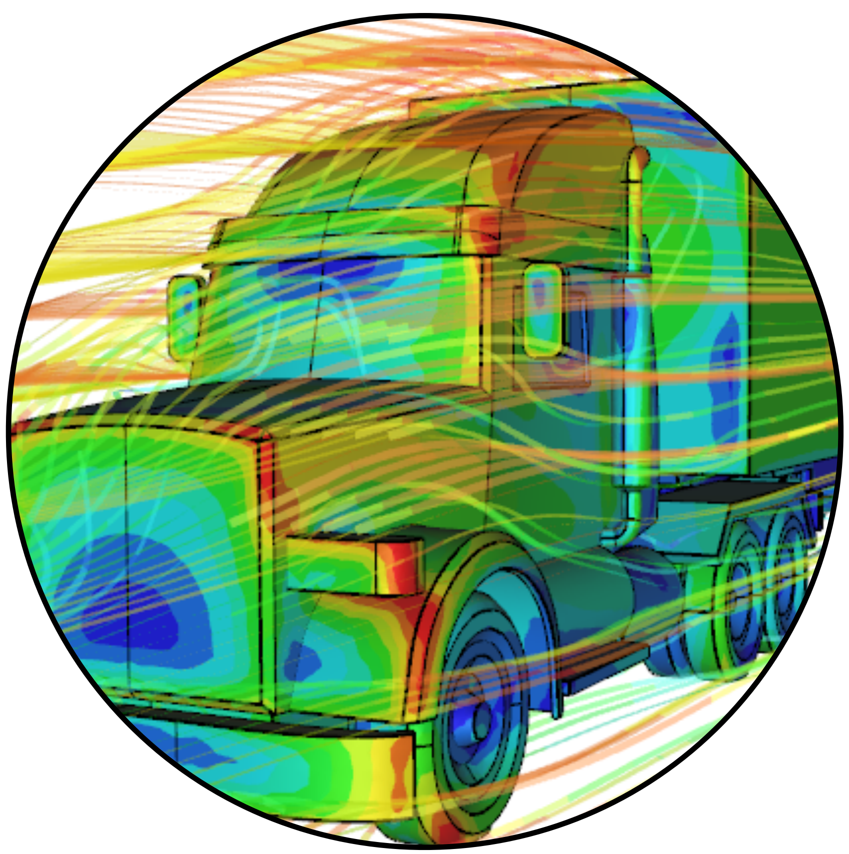
Goal
Solve for the quantity over time and space given its initial and boundary conditions.

Partial differential equations permeate our world

They lay at the heart of predictive modeling

$$\frac{\partial \mathbf{u}}{\partial t} = \mathscr{F}[t, \mathbf{x}, \mathbf{u}, \nabla_{\mathbf{x}}\mathbf{u}, \ldots]$$

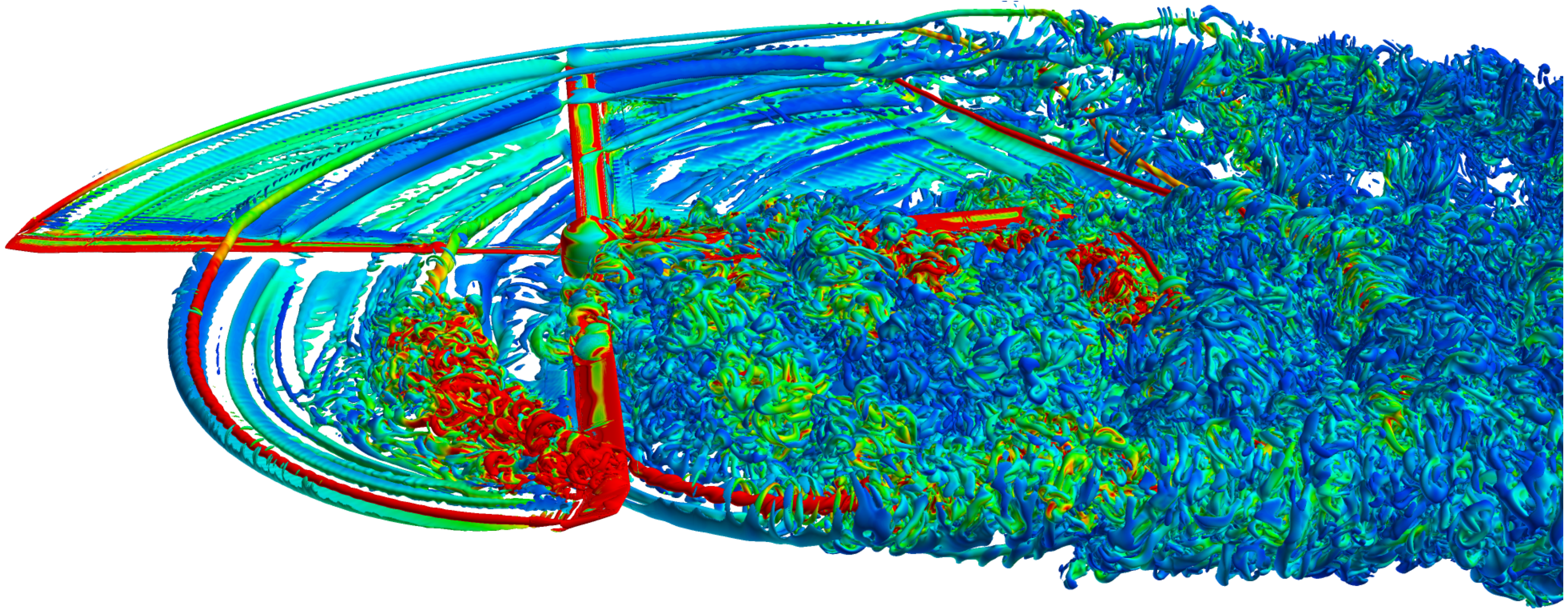
Engineering



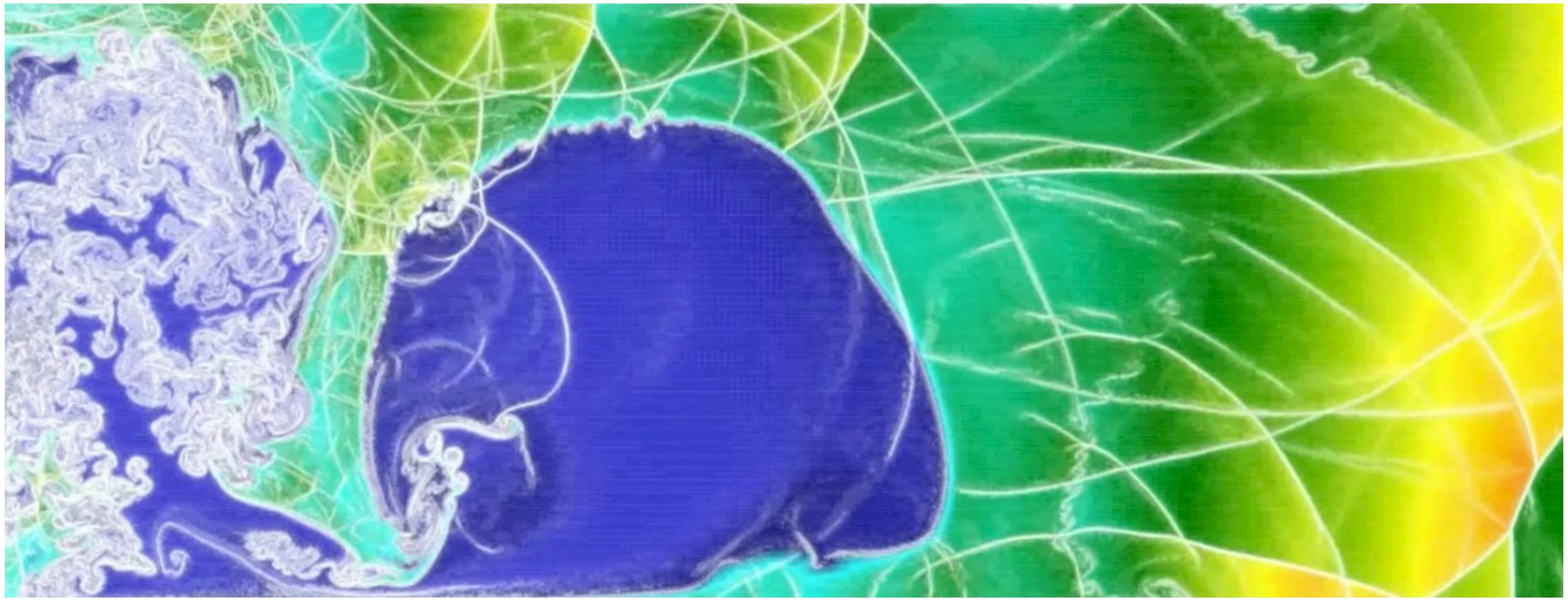
Physics

Finance

Modern numerical methods are impressive



Simulation of dynamic stall for a Blackhawk helicopter rotor in forward flight. (credit: NASA ARC).

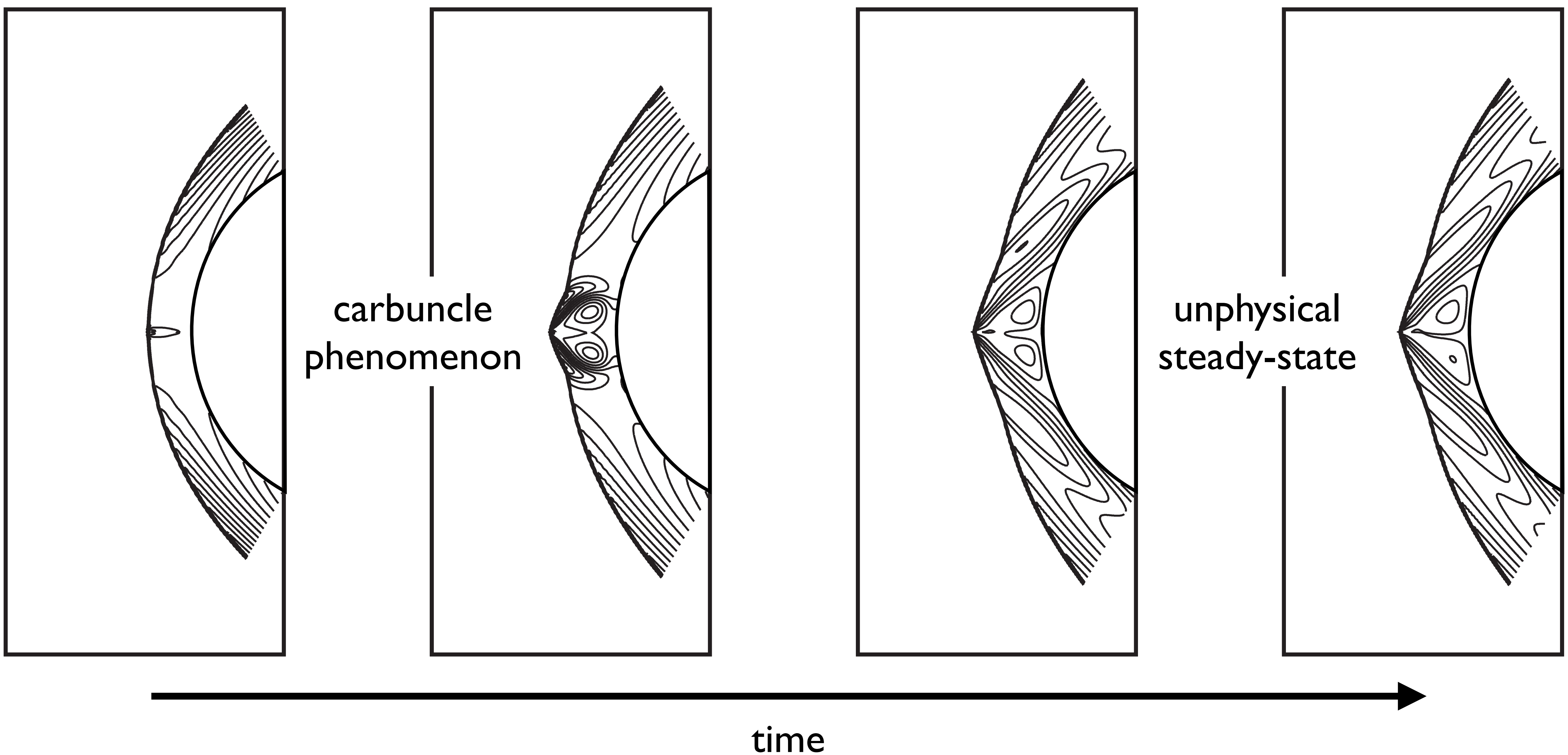


Simulation of ignition in a box. (credit: SpaceX in collaboration with Marc Massot of CMAP)

Challenges remain for many problems

Solution accuracy depends on mesh alignment and resolution

Water flow around circular pillar

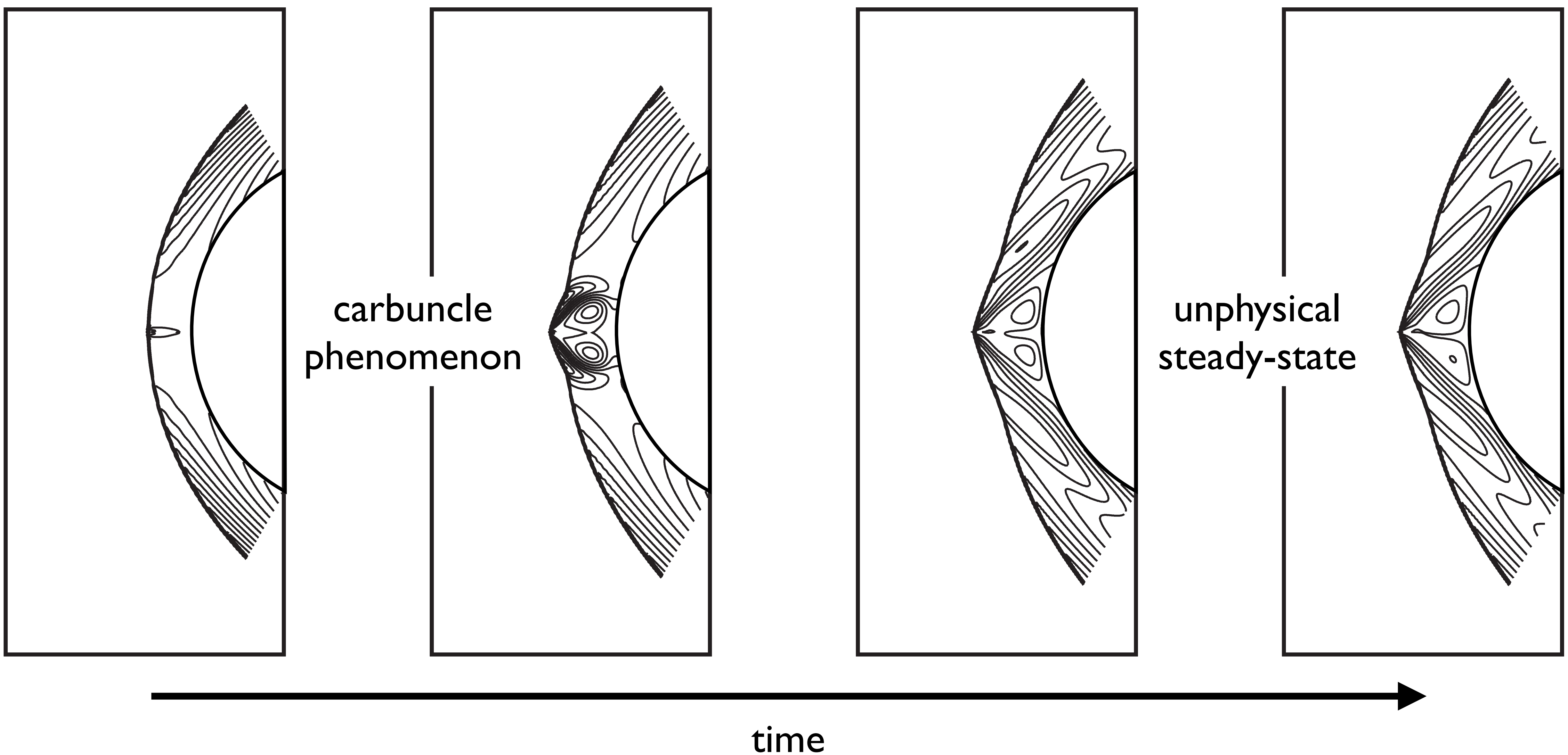


Friedemann Kemm, *App. Math. and Comp.* 320:596-613, 2018.

Challenges remain for many problems

Solution accuracy depends on mesh alignment and resolution

Water flow around circular pillar



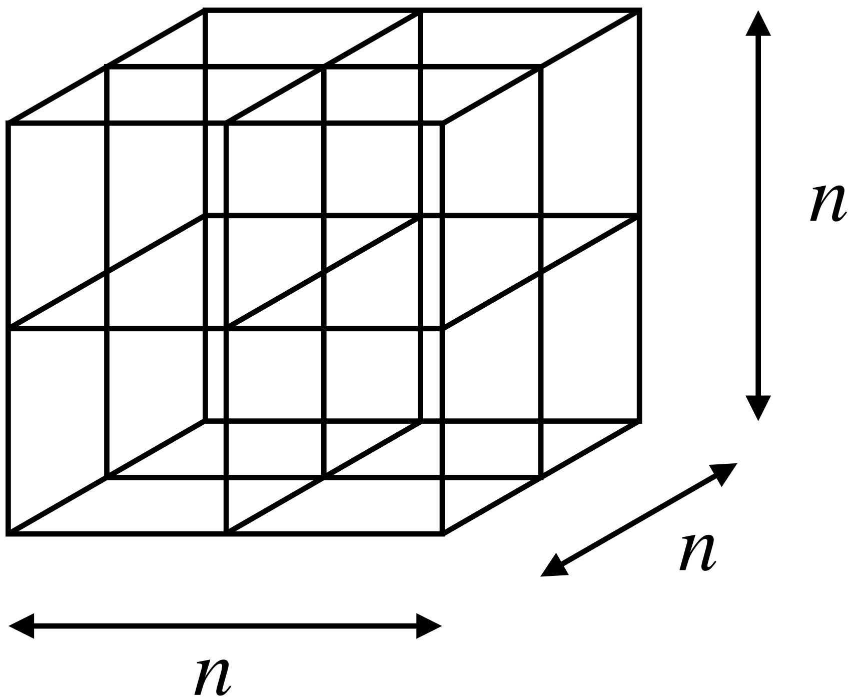
Mesh must be adapted to align with critical flow structures to maintain accuracy.

Friedemann Kemm, *App. Math. and Comp*. 320:596-613, 2018.

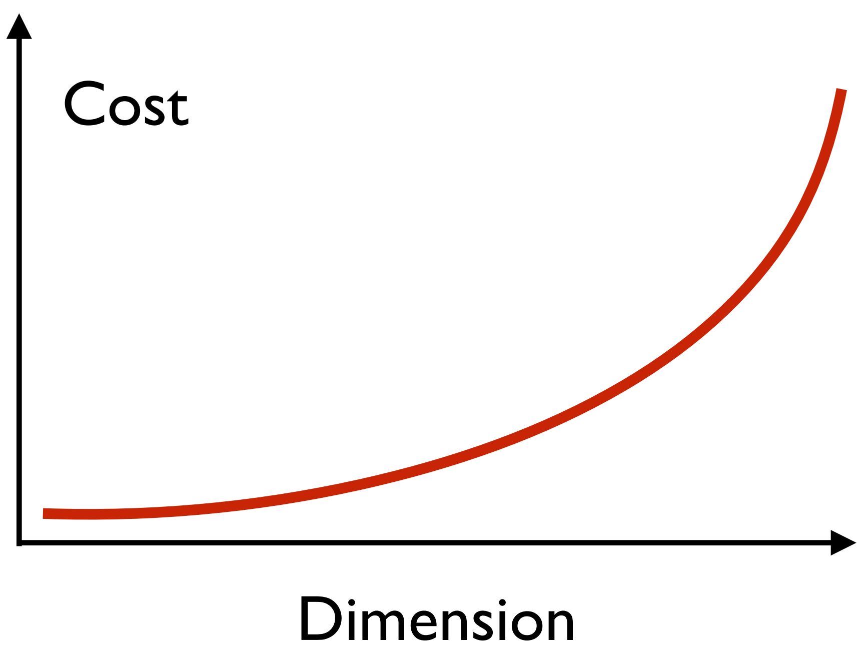
Challenges remain for many problems

Mesh size (and cost) scales exponentially with dimension

Number of cells = n^d



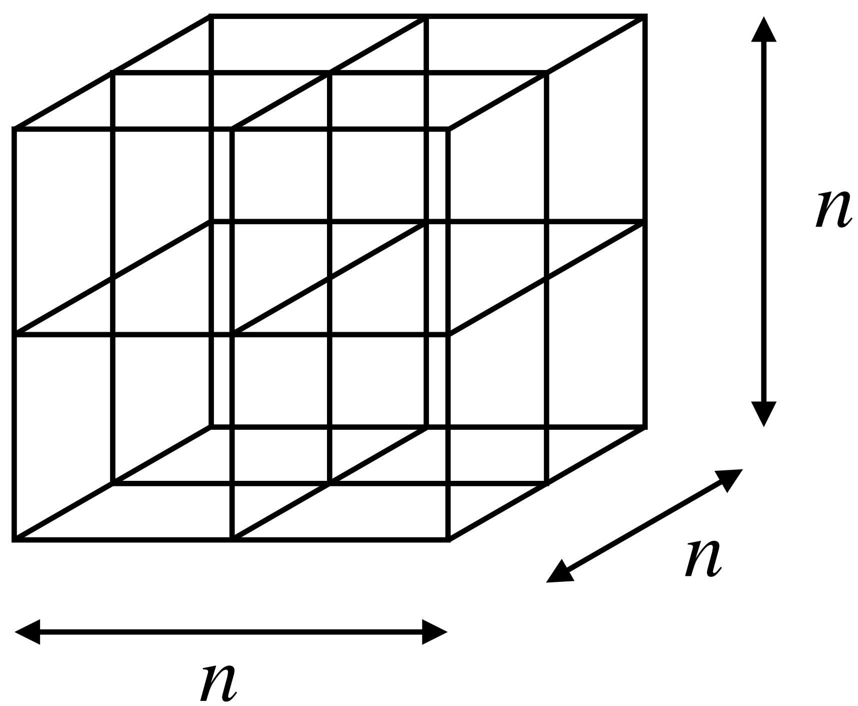
CPU Cost ∝ Number of Cells



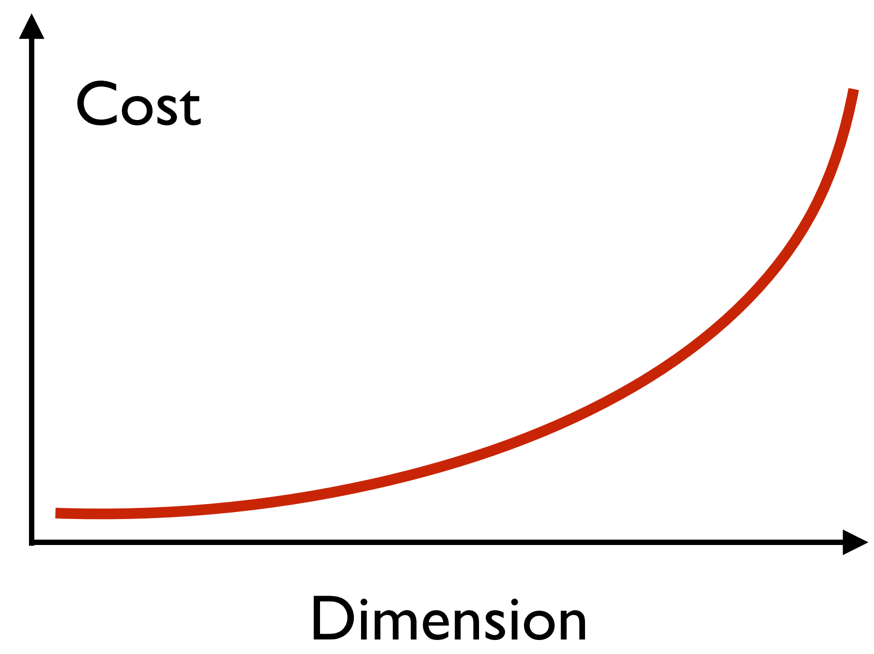
Challenges remain for many problems

Mesh size (and cost) scales exponentially with dimension

Number of cells = n^d



CPU Cost ∝ Number of Cells

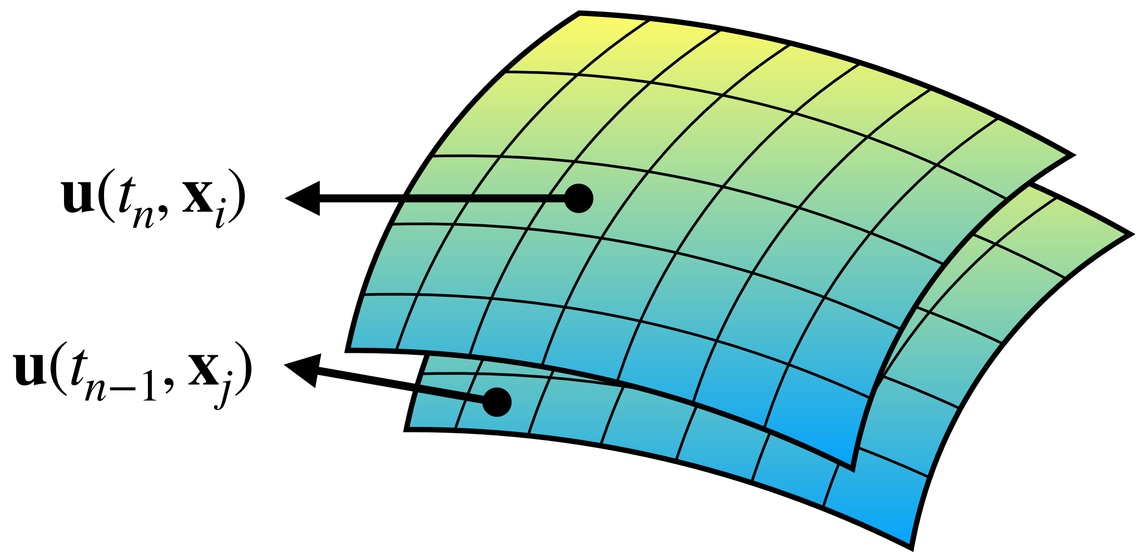


Curse of dimensionality: requires multi-resolution, high-order, or other schemes to solve complex problems in a reasonable amount of time.

Can we remove the mesh completely?

Can we remove the mesh completely?

Conventional Discretization Methods

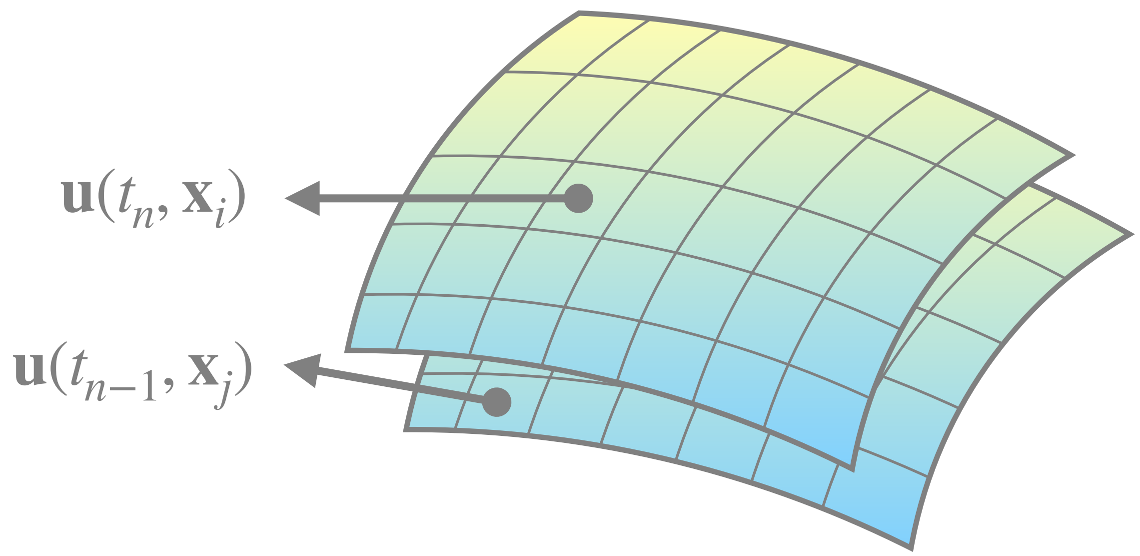


Problem converted to large system of ordinary differential equations

$$\frac{\partial \mathbf{u}_i}{\partial t} = F(\mathbf{u}_1, \ldots, \mathbf{u}_N)$$

Can we remove the mesh completely?

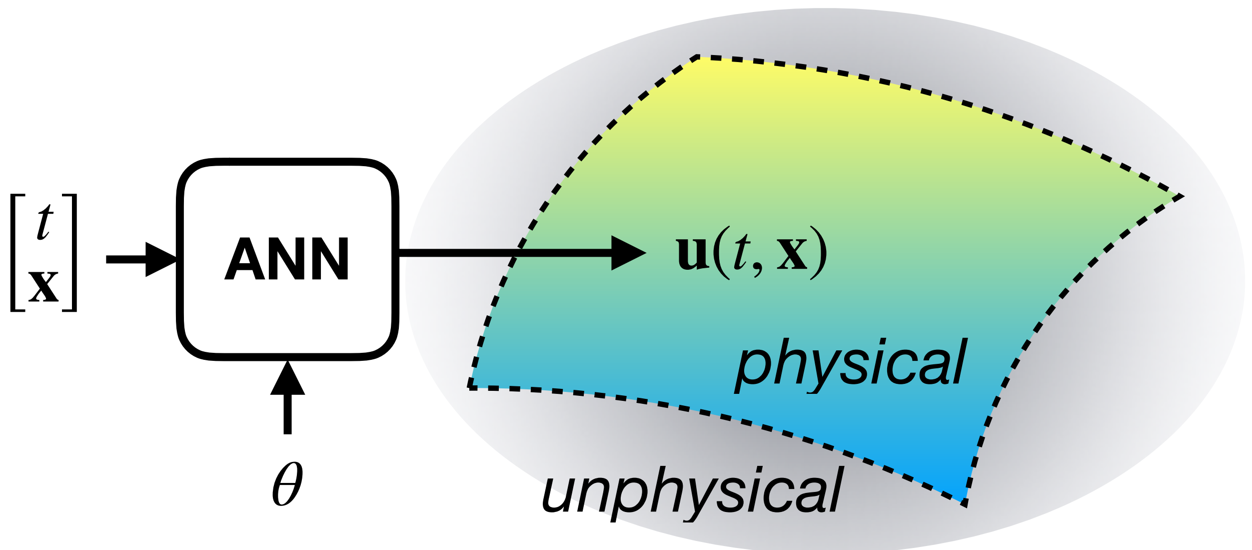
Conventional Discretization Methods



Problem converted to large system of ordinary differential equations

$$\frac{\partial \mathbf{u}_i}{\partial t} = F(\mathbf{u}_1, \ldots, \mathbf{u}_N)$$

Deep Learning Approach



Problem converted to optimization of neural network parameters.

$$\min_{\theta} \sum_{(t,x)_i} \left| \frac{\partial \mathbf{u}(\theta)}{\partial t} - \mathscr{F}[\mathbf{u}(\theta)] \right|$$

Neural Networks

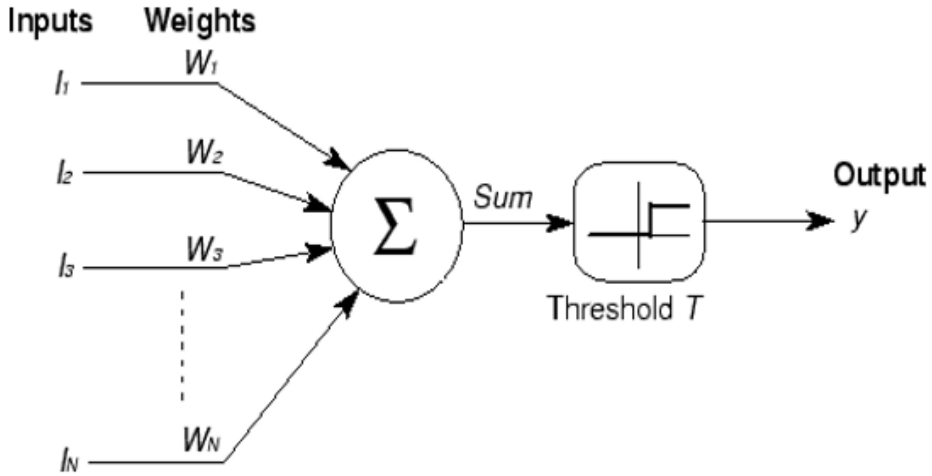
(Artificial) Neural Networks

Frank Rosenblatt developed first **perceptron** in 1958 to model the decision making of a fly.

F. Rosenblatt, *Psychological Review* 65(6):386-408, 1958.

(Artificial) Neural Networks

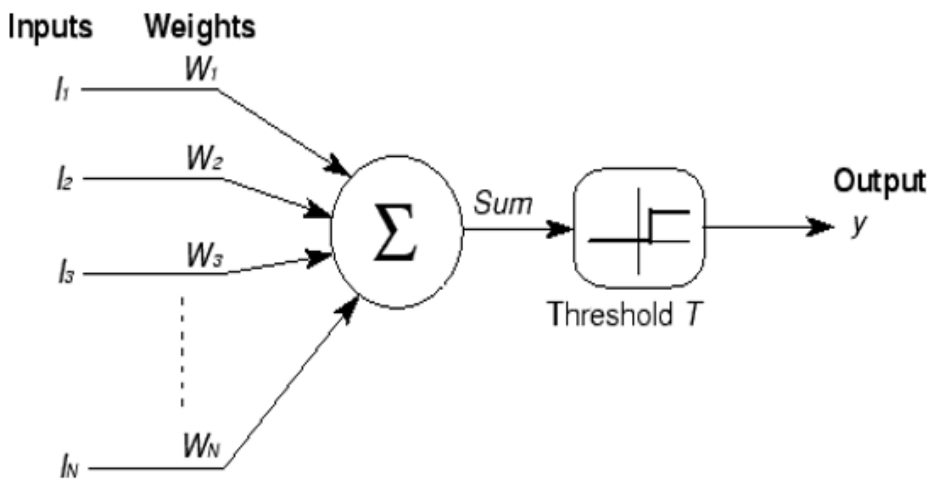
Frank Rosenblatt developed first **perceptron** in 1958 to model the decision making of a fly.

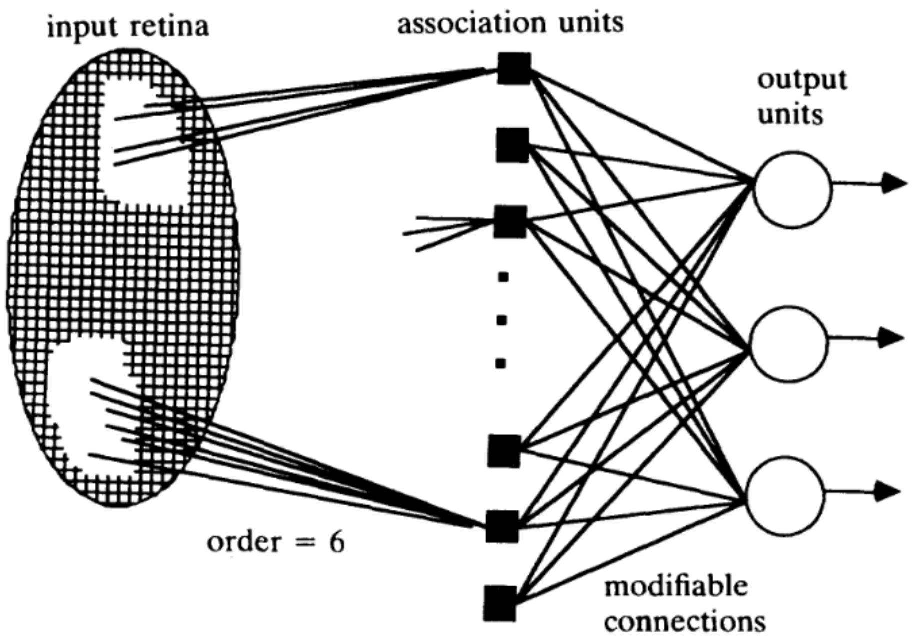


F. Rosenblatt, *Psychological Review* 65(6):386-408, 1958.

(Artificial) Neural Networks

Frank Rosenblatt developed first **perceptron** in 1958 to model the decision making of a fly.

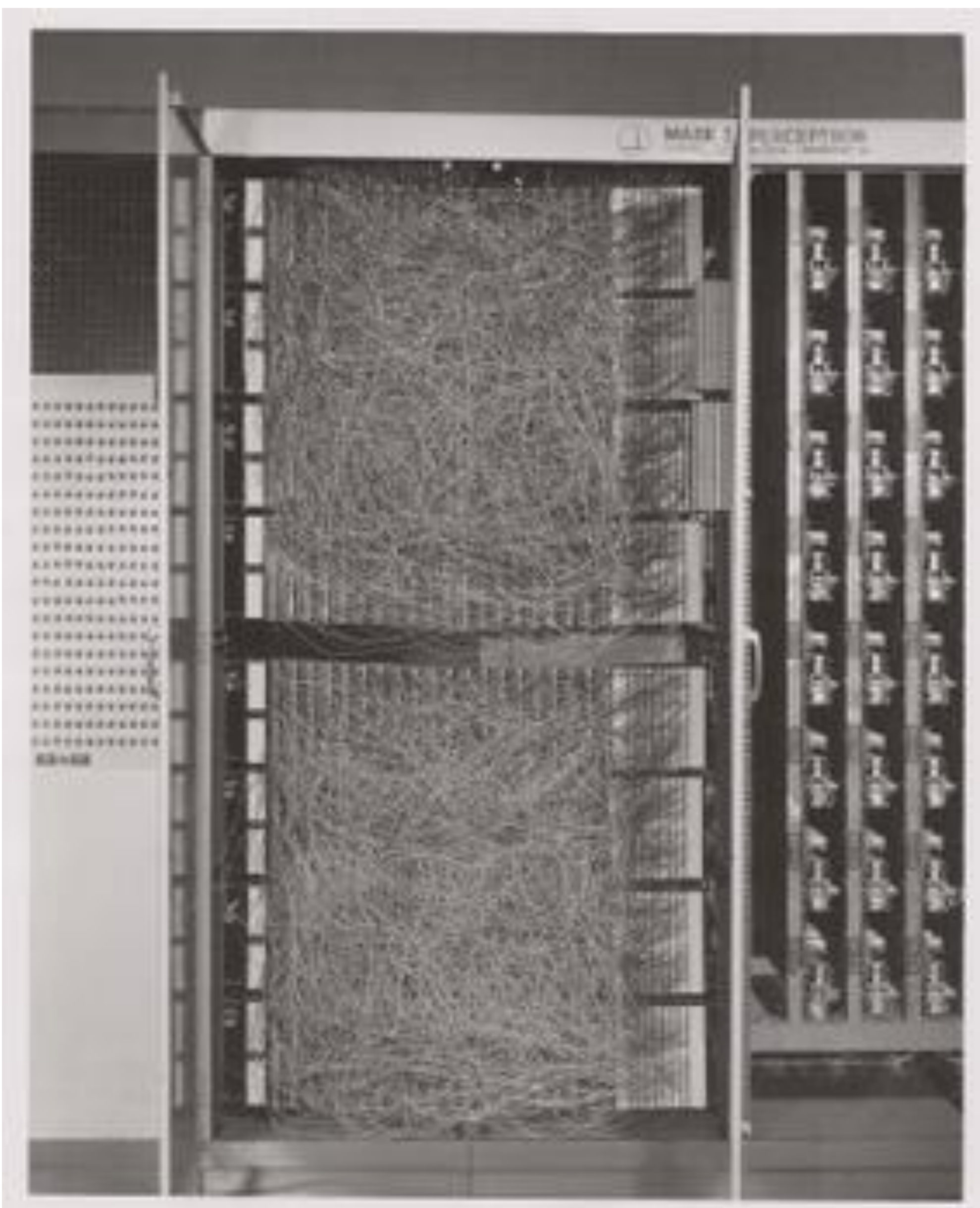


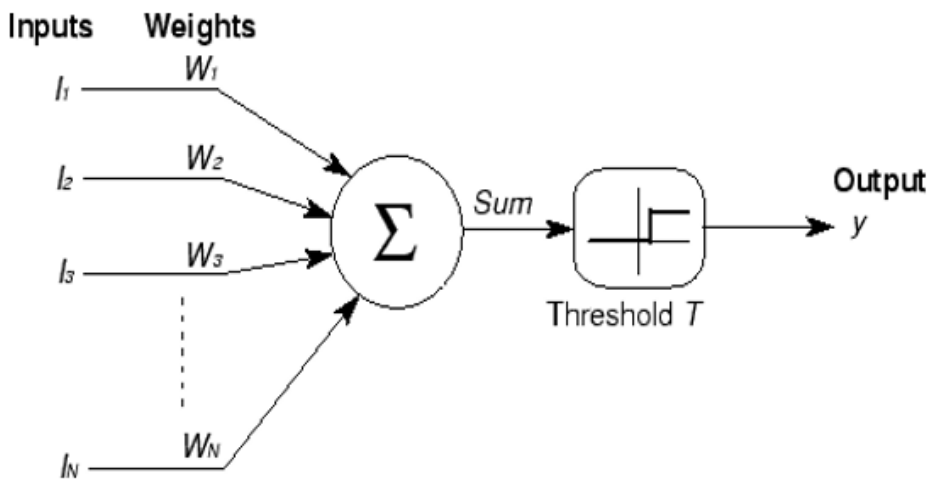


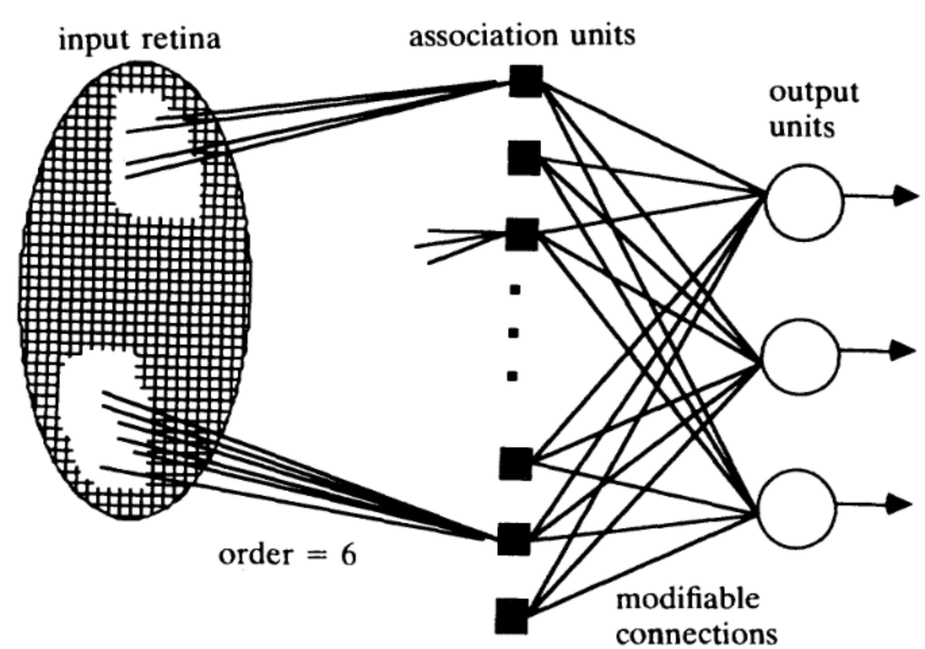
F. Rosenblatt, *Psychological Review* 65(6):386-408, 1958.

(Artificial) Neural Networks

Frank Rosenblatt developed first **perceptron** in 1958 to model the decision making of a fly.

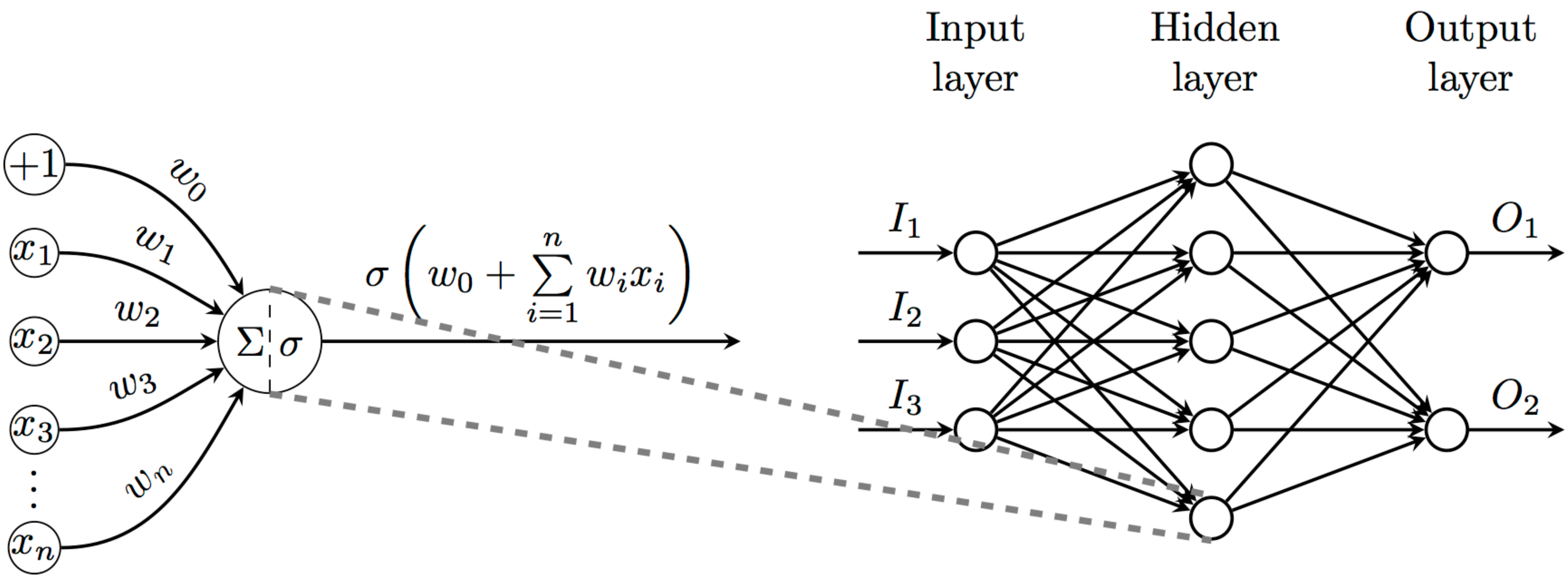






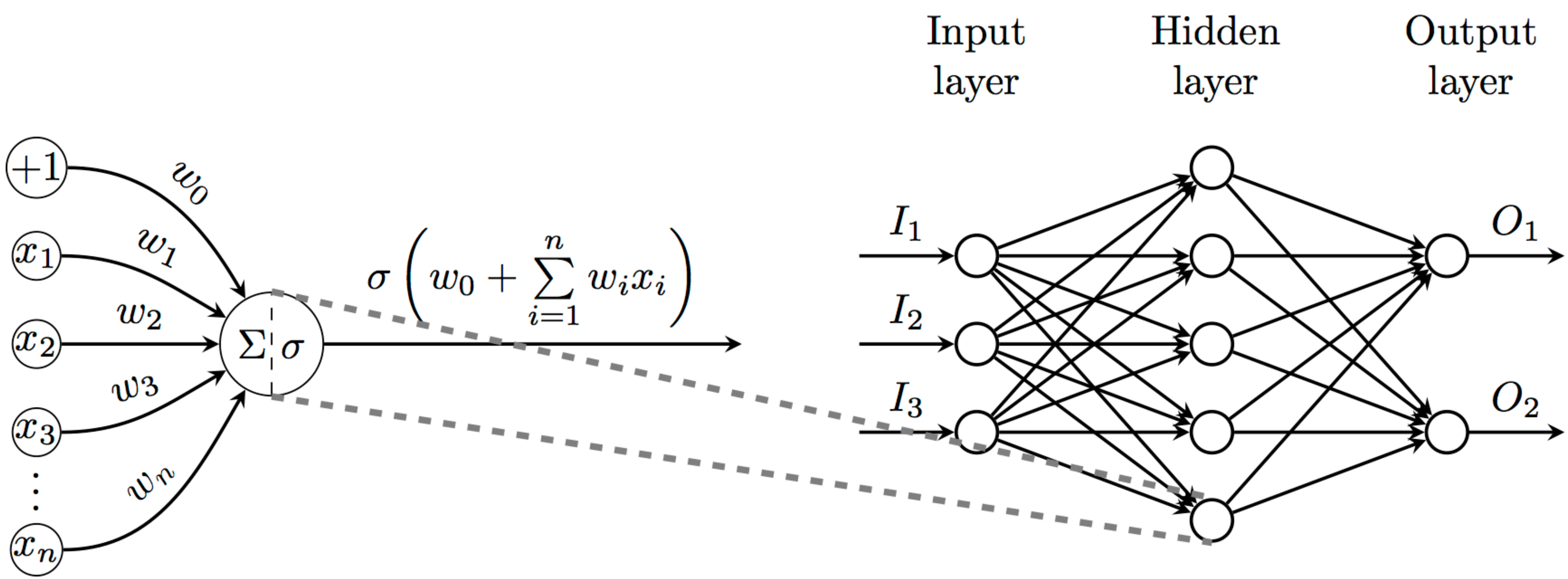
F. Rosenblatt, *Psychological Review* 65(6):386-408, 1958.

Multilayer Neural Networks



Credit: https://github.com/PetarV-

Multilayer Neural Networks



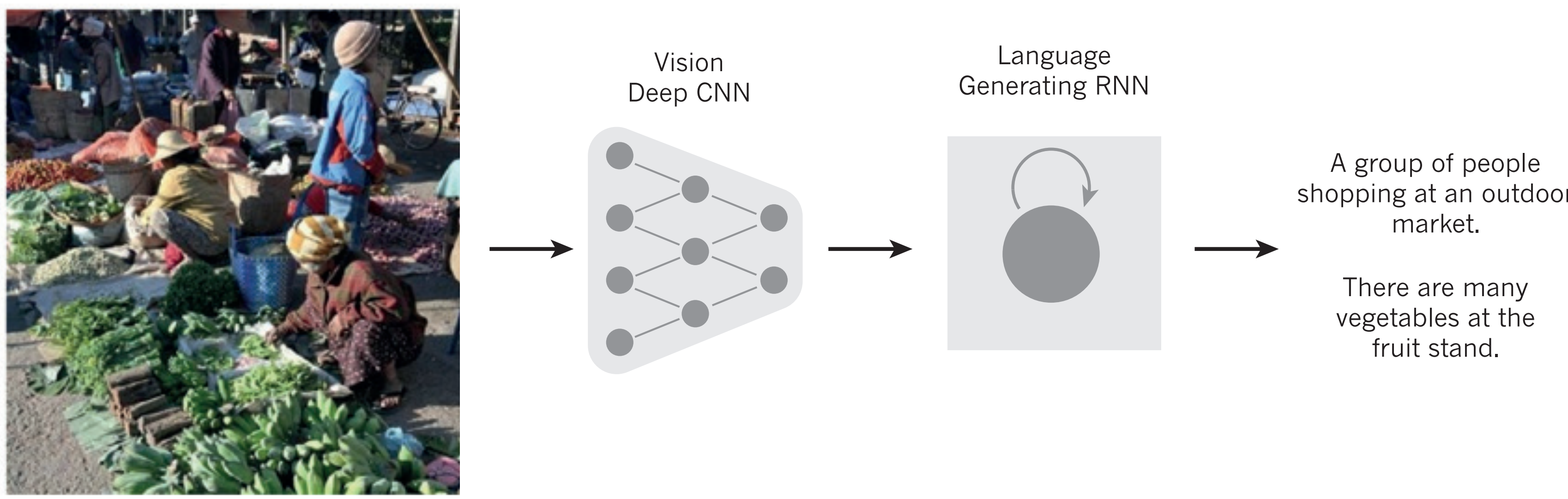
Credit: https://github.com/PetarV-

Universal Approximation Theorem: A standard multilayer feedforward network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy…

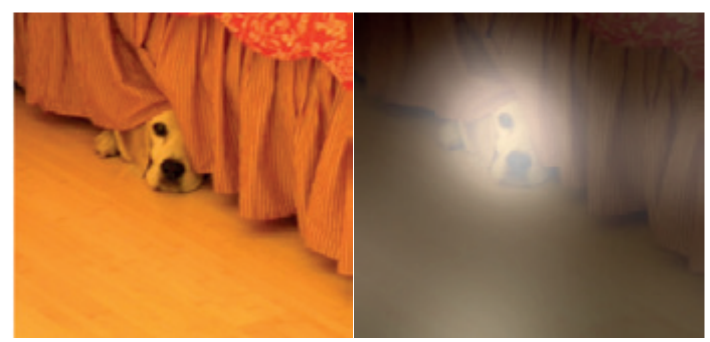
Leshno et al., Stern School of Business Working Paper Series STERN IS-92-13, 1992.

Modern networks leverage complex structure

Automatic image captioning

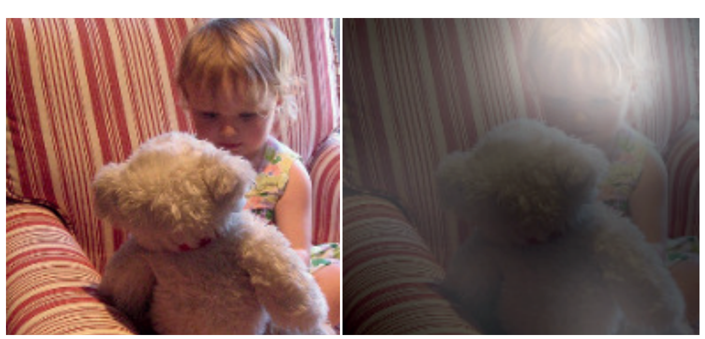


A woman is throwing a **frisbee** in a park.

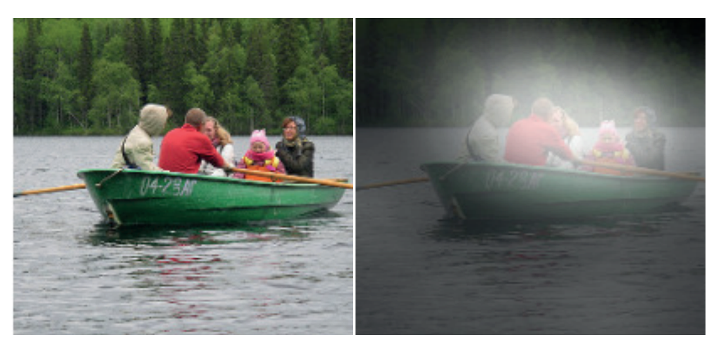


A **dog** is standing on a hardwood floor.

A **stop** sign is on a road with a mountain in the background



A little **girl** sitting on a bed with a teddy bear.



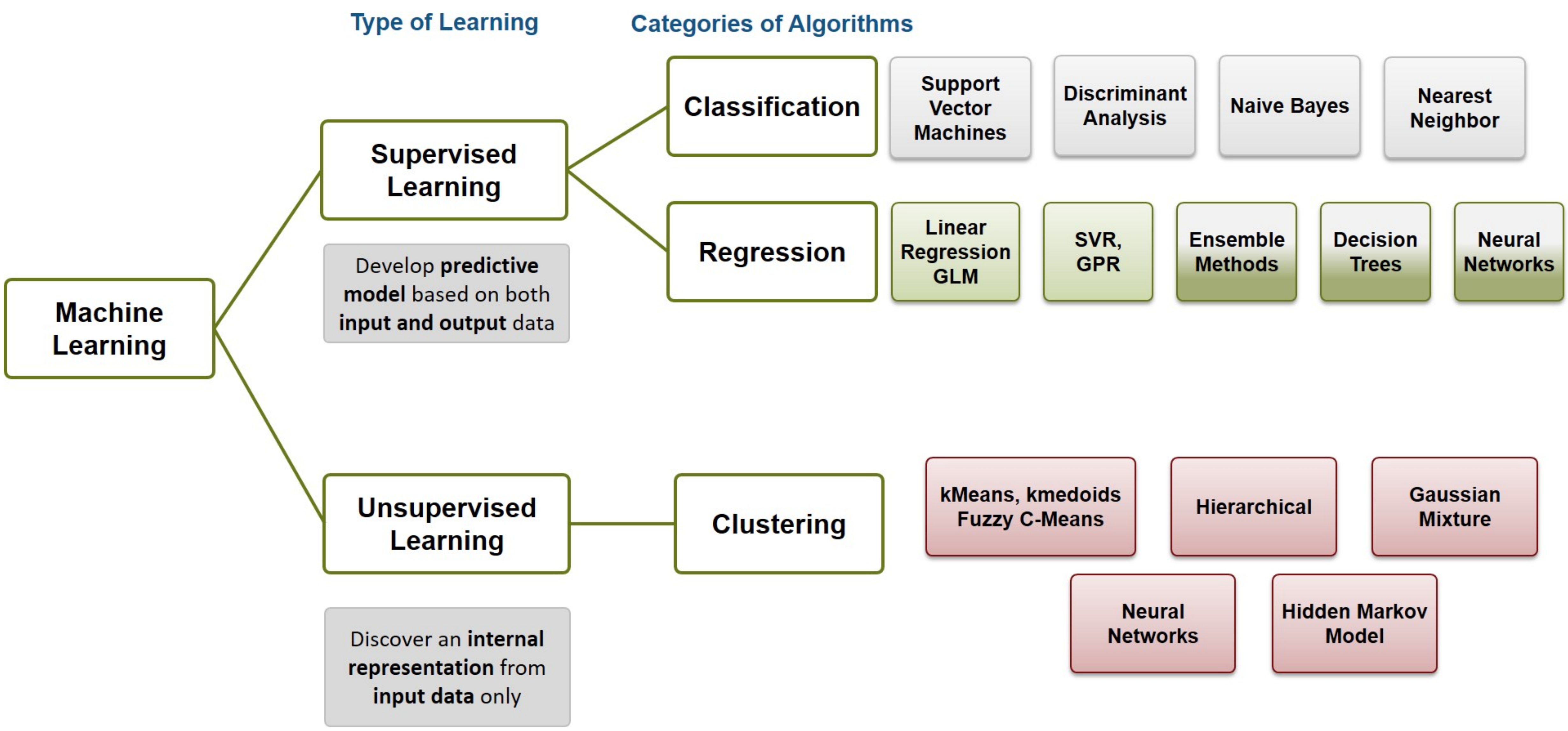
A group of **people** sitting on a boat in the water.

A giraffe standing in a forest with **trees** in the background.

LeCun, Bengio, Hinton. *Nature* 25:436-444, 2015.

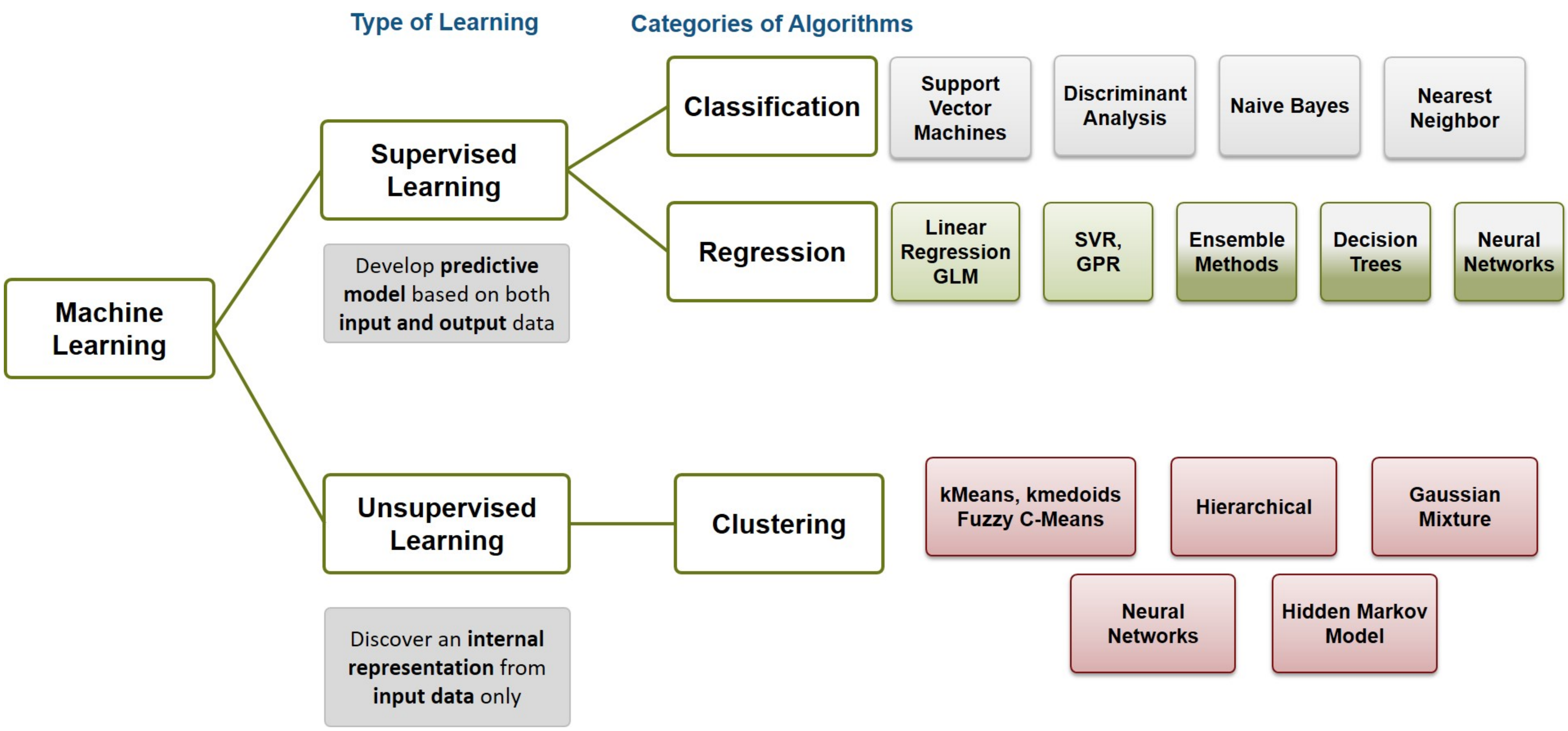
Learning

A network is said to **learn** if its **weights** are optimized against some **objective function**. In practice, this typically means that a **cost function** is minimized.



Learning

A network is said to **learn** if its **weights** are optimized against some **objective function**. In practice, this typically means that a **cost function** is minimized.



Deep Learning refers to training an ANN with many hidden layers, the network is deep.

(Stochastic) Gradient Descent

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \dots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \ldots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}),\ \dots\ ,(\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

Define a cost function: $\mathscr{L} \ = \ \frac{1}{n}\sum_{i=1}^{n} l_i \ = \ \frac{1}{n}\sum_{i=1}^{n} \left\| f(\mathbf{X_i}\,;\,\theta) - \mathbf{Y_i} \right\|_2^2$

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \dots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

Define a cost function: $\mathscr{L} \;=\; \frac{1}{n}\sum_{i=1}^{n} l_i \;=\; \frac{1}{n}\sum_{i=1}^{n} \left\| f(\mathbf{X_i}\,;\,\theta) - \mathbf{Y_i} \right\|_2^2$

Minimize cost function: $\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathscr{L}$

(Stochastic) Gradient Descent

Given training data: $\mathcal{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \dots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

Define a cost function: $\mathcal{L} \ = \ \frac{1}{n}\sum_{i=1}^{n} l_i \ = \ \frac{1}{n}\sum_{i=1}^{n} \left\| f(\mathbf{X_i}\,;\,\theta) - \mathbf{Y_i} \right\|_2^2$

Minimize cost function: $\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}$

Algorithm:

1. Initialize weights $\theta^0 = \mathcal{N}(0, \mu)$
2. Update based on gradient $\theta^{k+1} = \theta^k - \lambda\, \nabla_\theta \mathcal{L}$
3. Repeat until convergence $\lim_{k \to \infty} \theta^k = \theta^*$

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \ldots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

Define a cost function: $\mathscr{L} \ = \ \frac{1}{n}\sum_{i=1}^{n} l_i \ = \ \frac{1}{n}\sum_{i=1}^{n} \left\| f(\mathbf{X_i}\,;\,\theta) - \mathbf{Y_i} \right\|_2^2$

Minimize cost function: $\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathscr{L}$

Algorithm:

1. Initialize weights $\theta^0 = \mathcal{N}(0, \mu)$
2. Update based on gradient $\theta^{k+1} = \theta^k - \lambda \, \nabla_\theta \mathscr{L}$
3. Repeat until convergence $\lim_{k \to \infty} \theta^k = \theta^*$

Convergence is guaranteed if cost function is convex. (and normally if it isn't)

(Stochastic) Gradient Descent

Given training data: $\mathscr{D}_n = \{(\mathbf{X_1}, \mathbf{Y_1}), \ \dots \ , (\mathbf{X_n}, \mathbf{Y_n})\}$

Define neural network: $f(\mathbf{X}\,;\,\theta) \mapsto \hat{\mathbf{Y}}$

Define a cost function: $\mathscr{L} \ = \ \frac{1}{n}\sum_{i=1}^{n} l_i \ \approx \ \frac{1}{|\,\mathscr{I}\,|}\sum_{i\in\mathscr{I}} \left\| f(\mathbf{X_i}\,;\,\theta) - \mathbf{Y_i} \right\|_2^2$

Minimize cost function: $\theta^* = \underset{\theta}{\operatorname{argmin}}\,\mathscr{L}$

Algorithm:

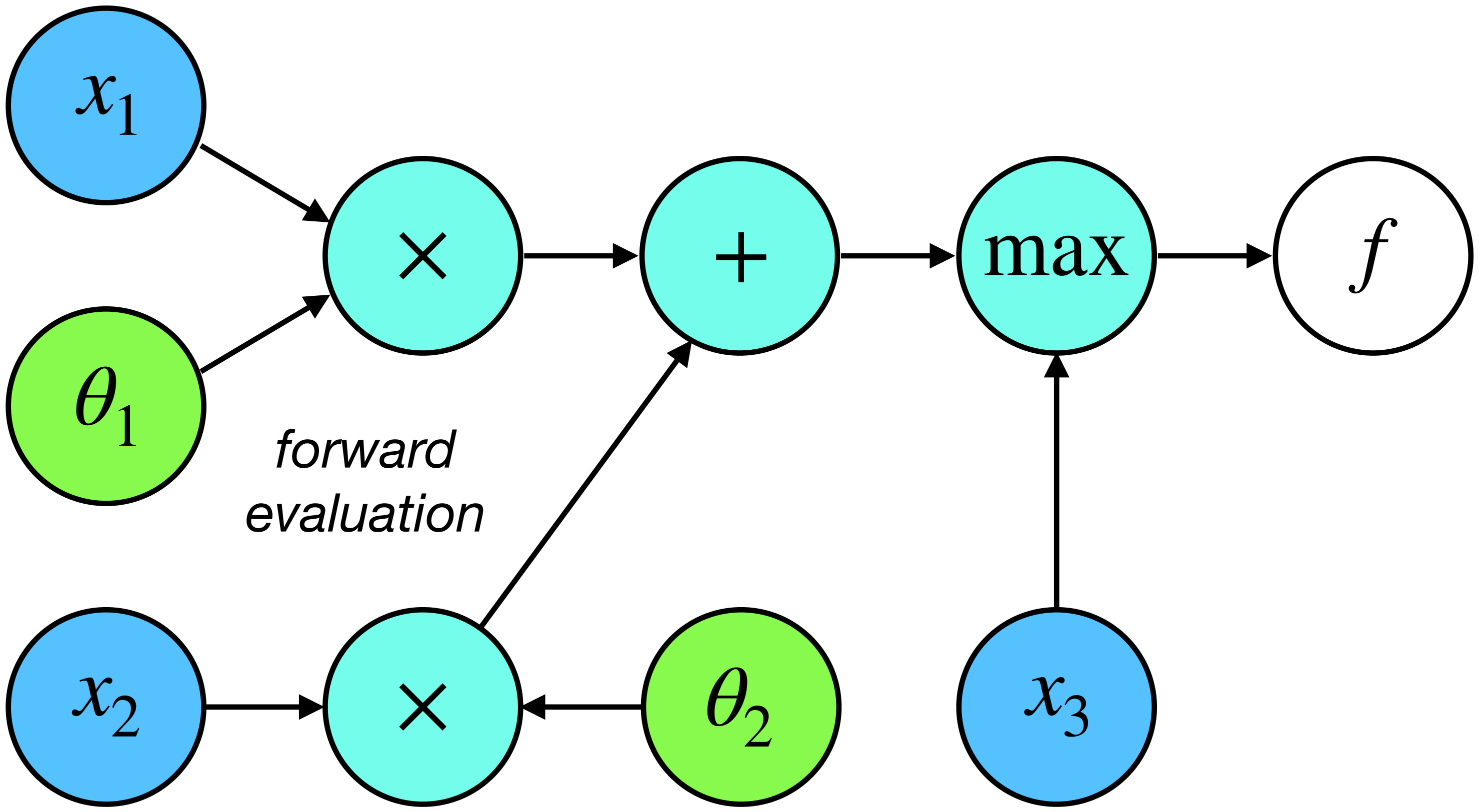
1. Initialize weights $\theta^0 = \mathscr{N}(0, \mu)$
2. Update based on gradient $\theta^{k+1} = \theta^k - \lambda\,\nabla_\theta \mathscr{L}$
3. Repeat until convergence $\lim_{k\to\infty} \theta^k = \theta^*$

Convergence is guaranteed if cost function is convex. (and normally if it isn't)

How do we get the gradient?

Consider the **computational graph** for the simple function

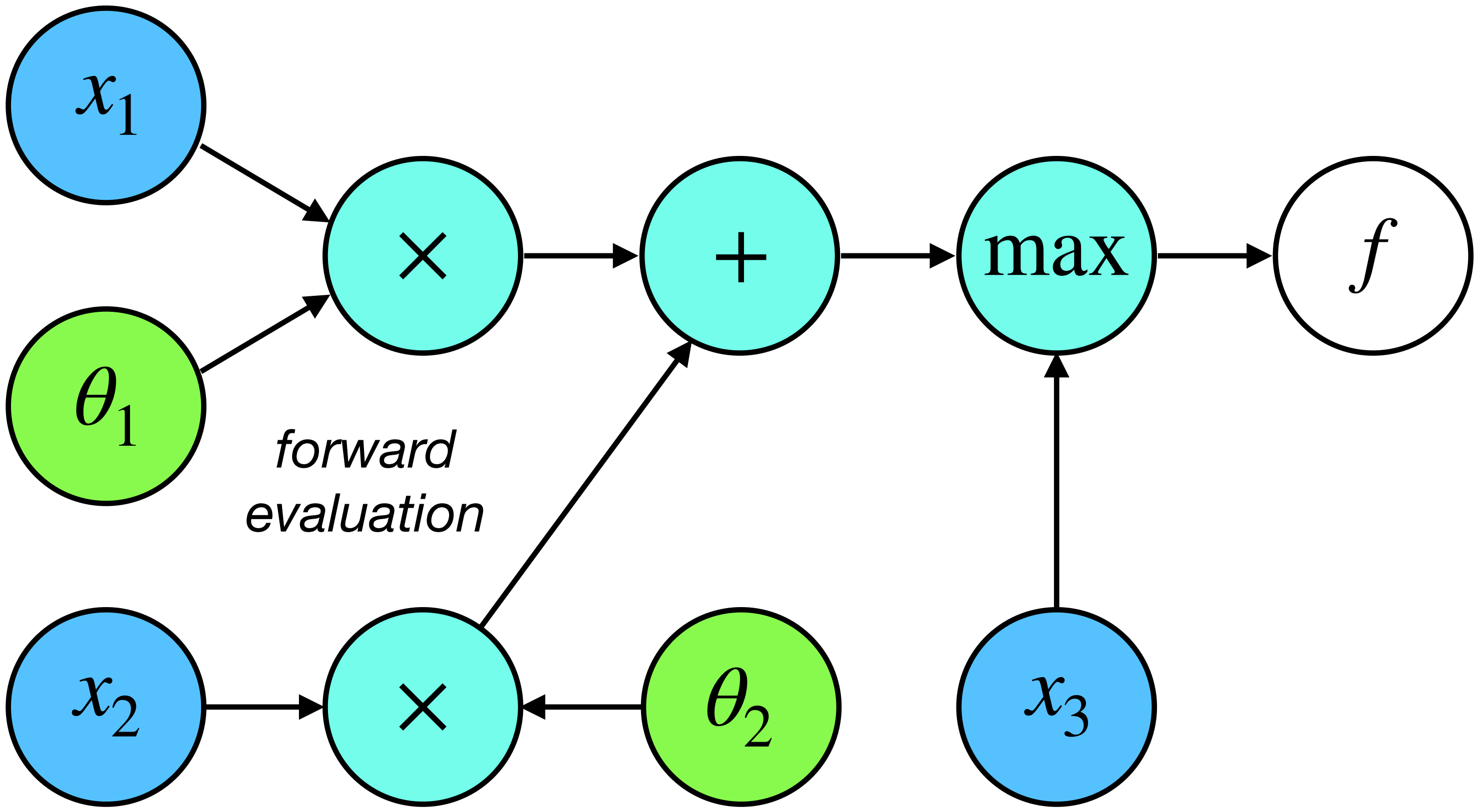
$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$



How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

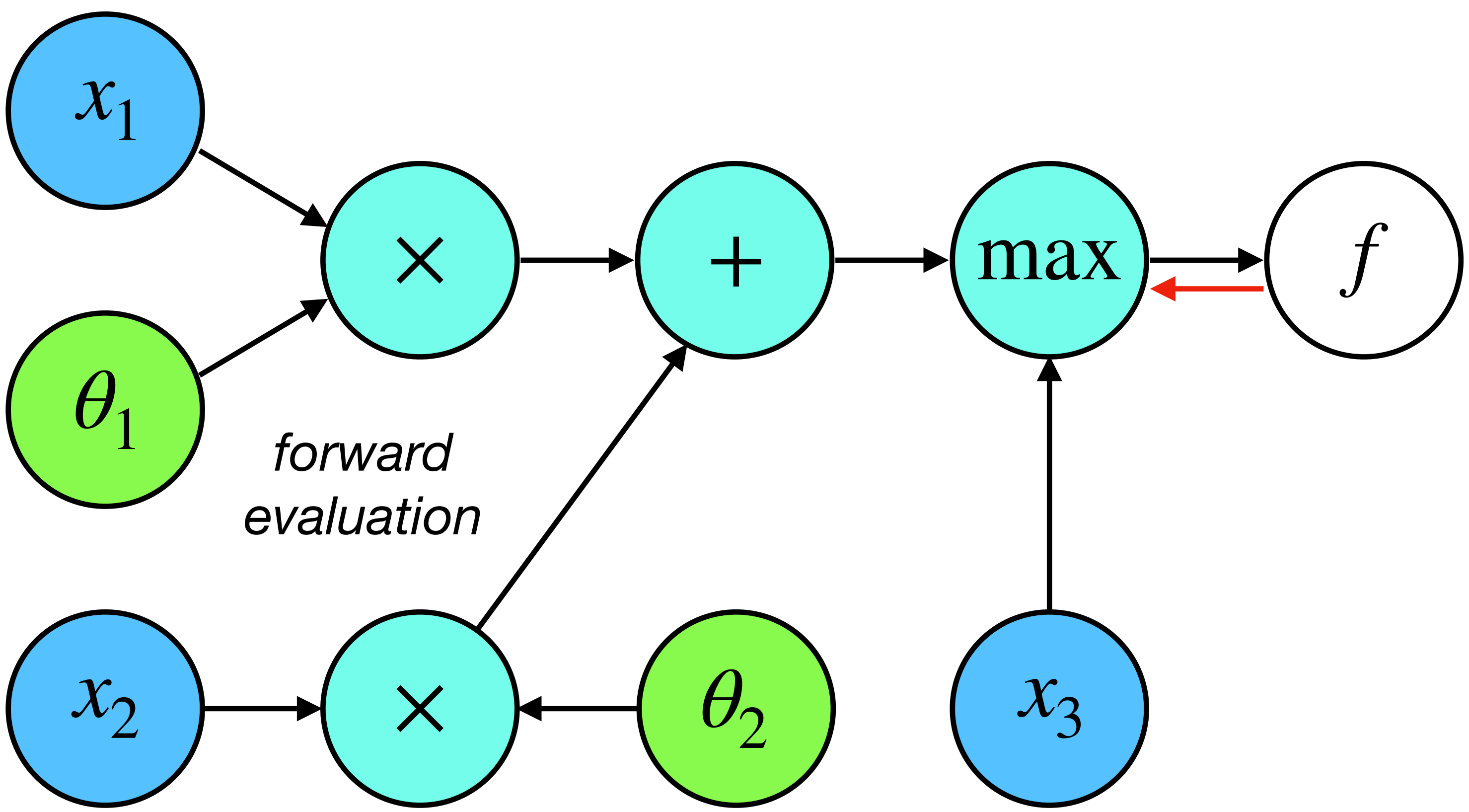


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

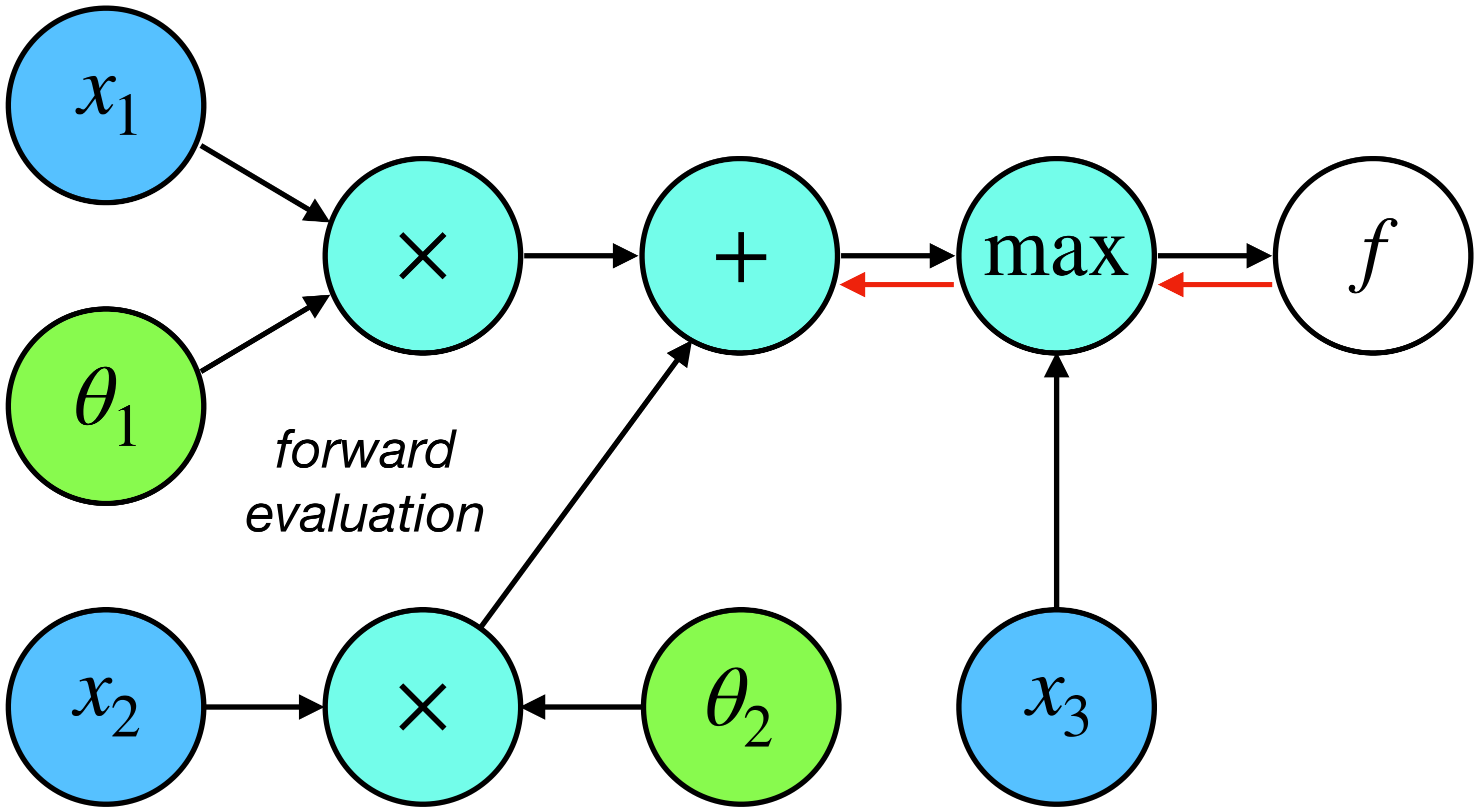


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

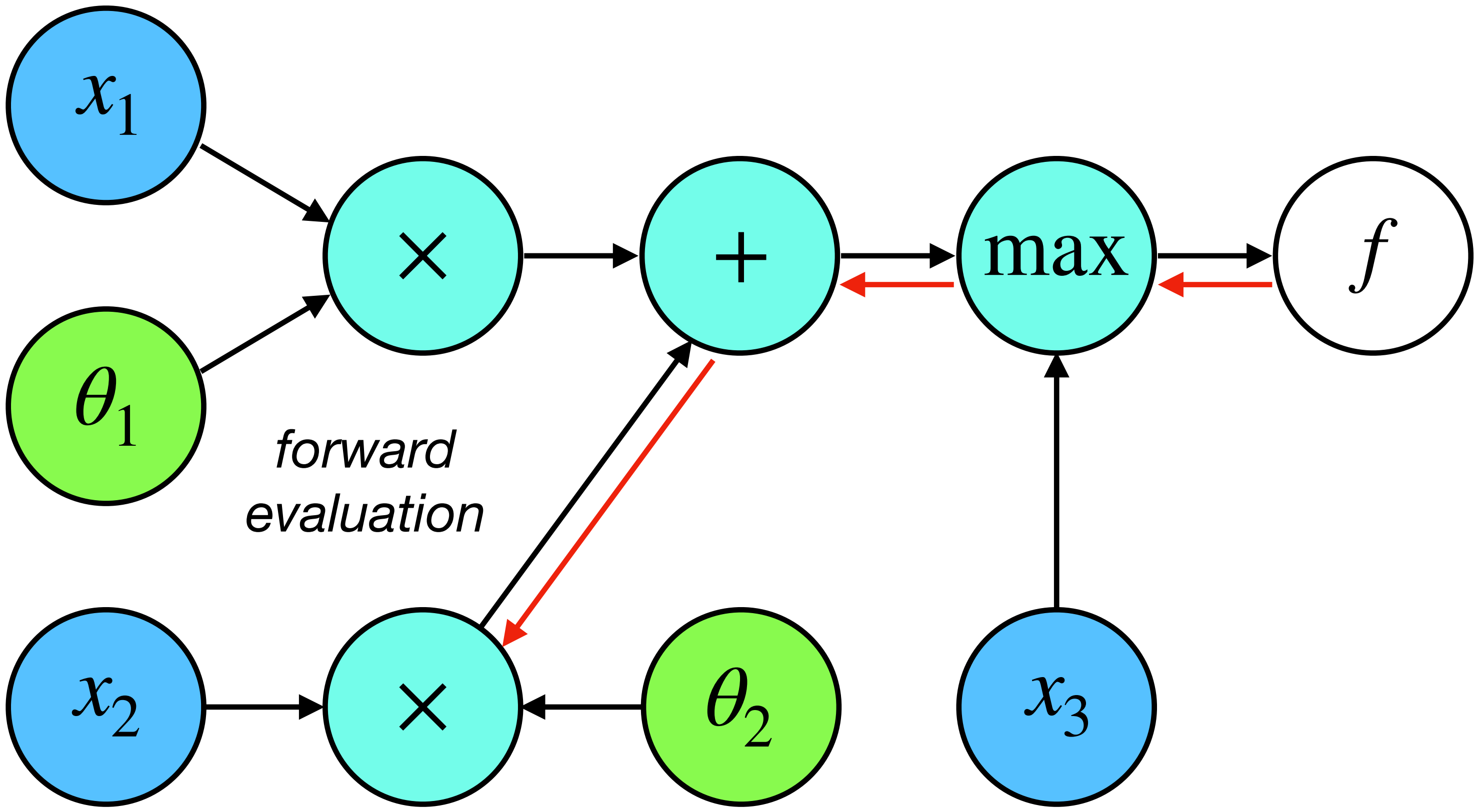


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

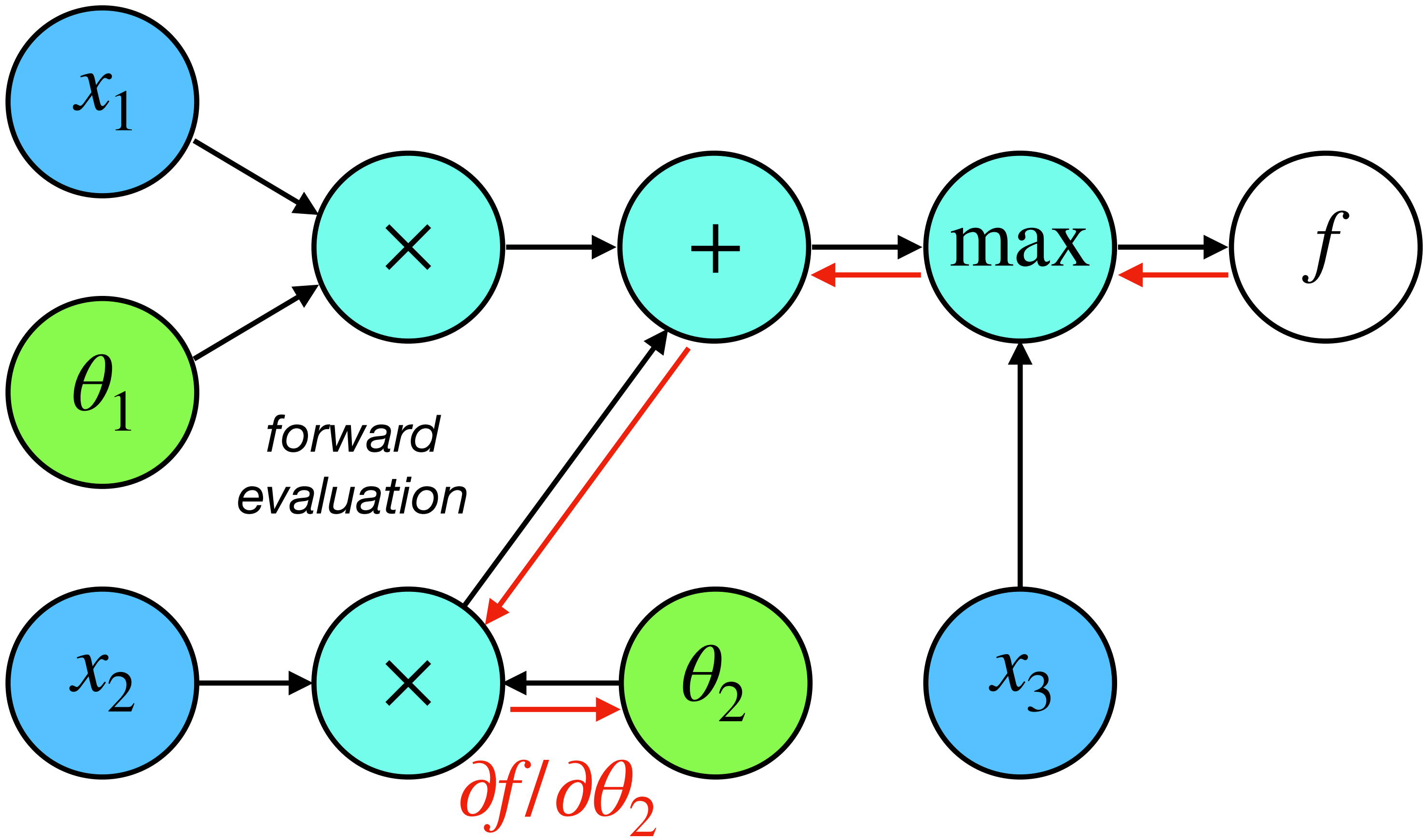


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

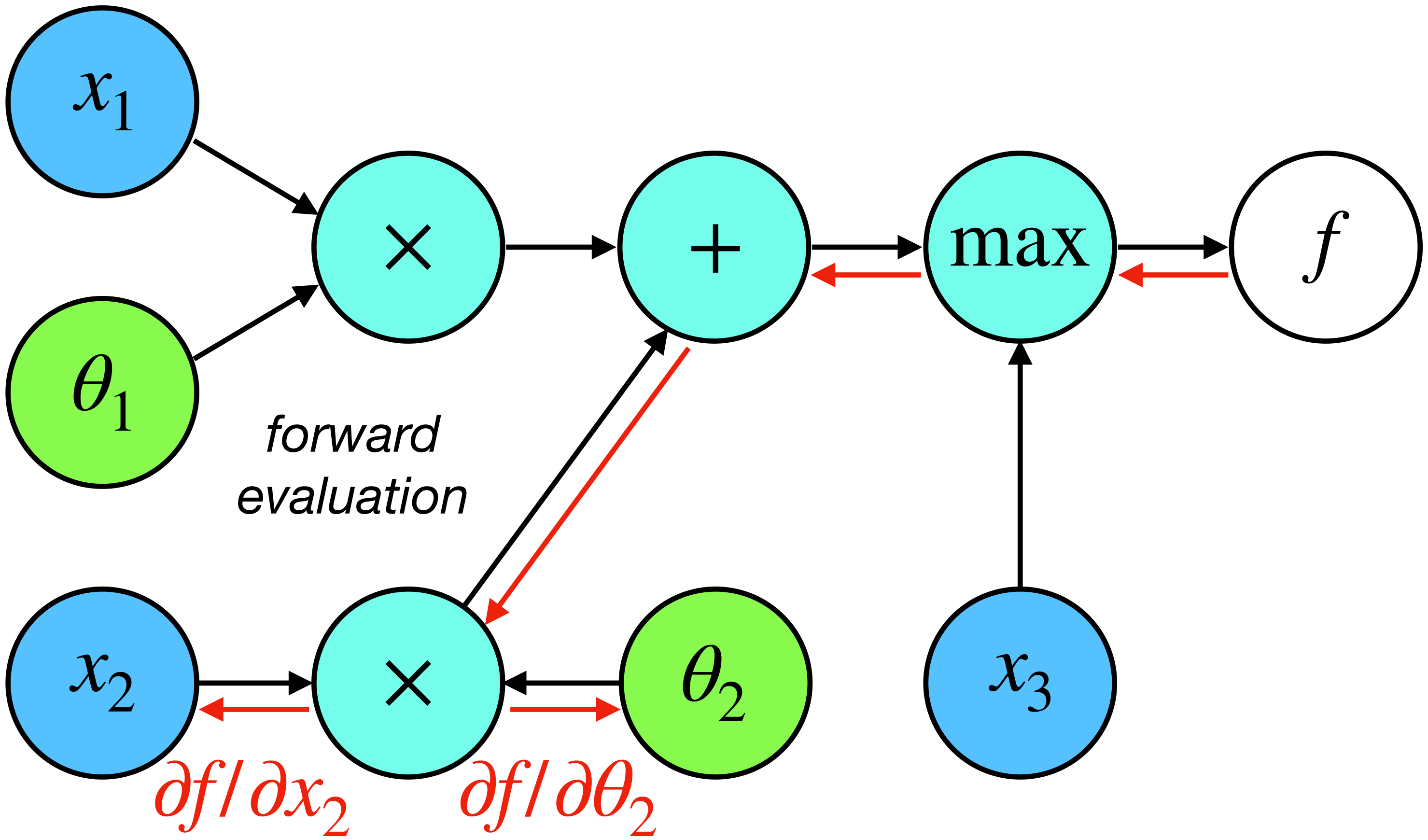


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

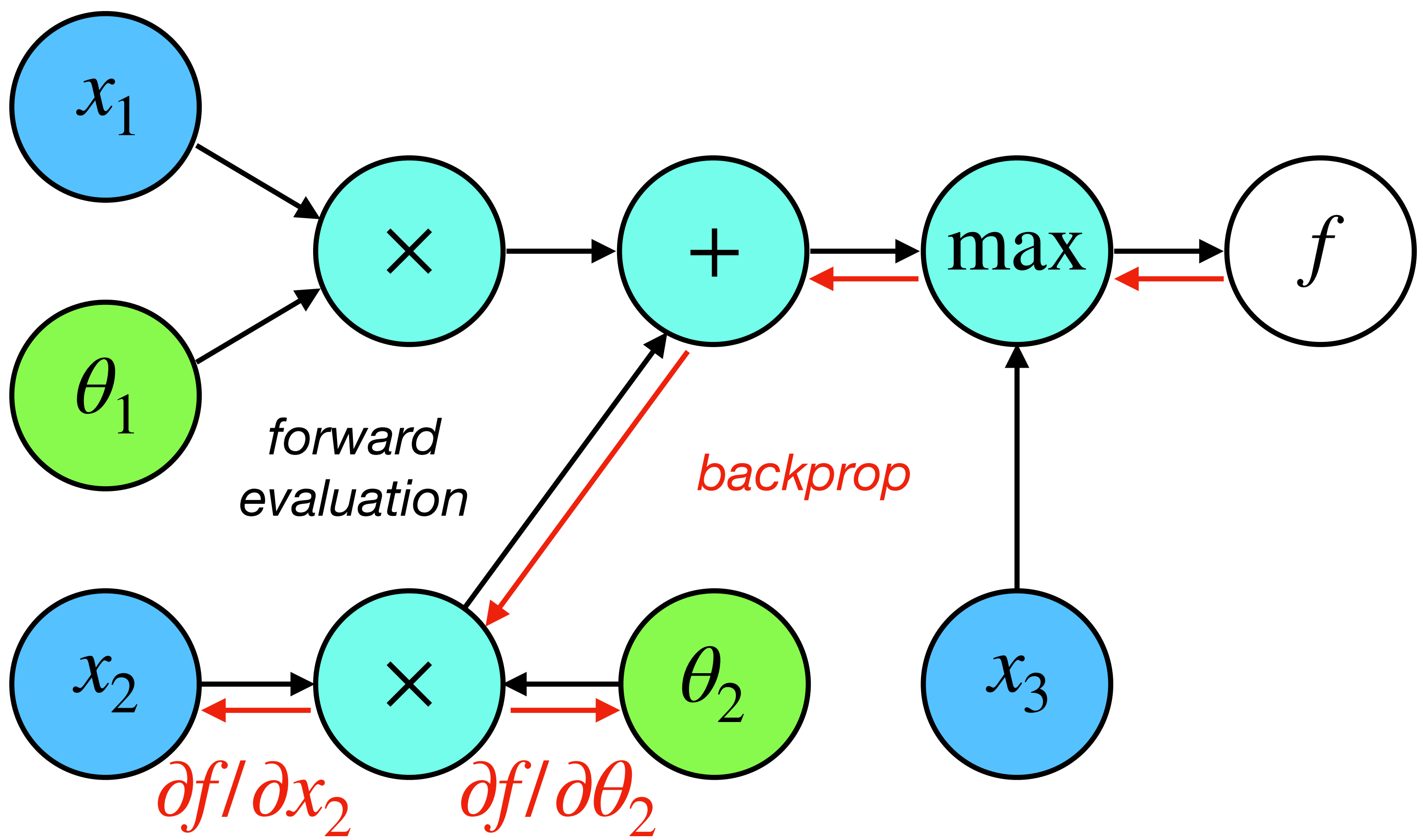


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$

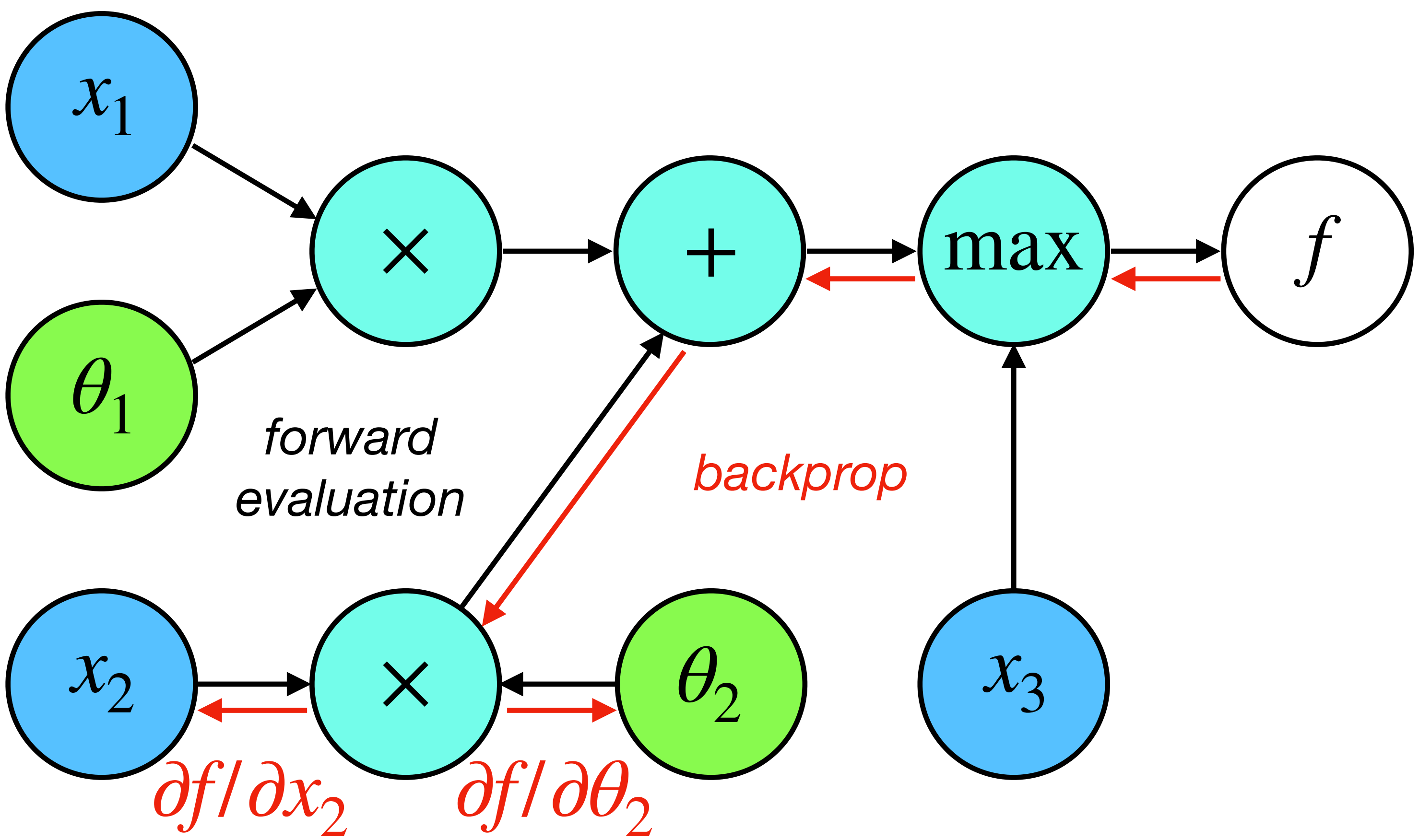


Gradient calculation through recursive uses of the chain rule.

How do we get the gradient?

Consider the **computational graph** for the simple function

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$



Gradient calculation through recursive uses of the chain rule.

Modern deep learning libraries implement NNs as computational graphs and provide functions to compute their gradients **analytically** with respect to any node in the graph, using **back-propagation**.

So how do we learn equations?

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \ldots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \dots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathcal{N}(x;\, \theta) \mapsto u$$

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \ldots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathcal{N}(x;\, \theta) \mapsto u$$

It turns out, that we can easily differentiate a neural network, and the **derivative is another network which shares the same parameters as the original**. Remember back-propagation!

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathscr{G}(x, u, \nabla u, \nabla^2 u, \ldots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathscr{N}(x;\, \theta) \mapsto u$$

It turns out, that we can easily differentiate a neural network, and the **derivative is another network which shares the same parameters as the original**. Remember back-propagation!

Replace the PDE with a neural network.

$$\mathscr{G}(x, \mathscr{N}, \nabla \mathscr{N}, \nabla^2 \mathscr{N}, \ldots;\, \theta) = 0$$

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \ldots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathcal{N}(x;\, \theta) \mapsto u$$

It turns out, that we can easily differentiate a neural network, and the **derivative is another network which shares the same parameters as the original**. Remember back-propagation!

Replace the PDE with a neural network.

$$\mathcal{G}(x, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \ldots;\, \theta) = 0$$

Now we have an optimization problem that we know how to solve.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i \in \mathcal{P}} \left\| \mathcal{G}(x_i, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \ldots;\, \theta) \right\|_2^2$$

So how do we learn <u>equations</u>?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \dots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathcal{N}(x;\, \theta) \mapsto u$$

It turns out, that we can easily differentiate a neural network, and the **derivative is another network which shares the same parameters as the original**. Remember back-propagation!

Replace the PDE with a neural network.

$$\mathcal{G}(x, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \dots;\, \theta) = 0$$

Now we have an optimization problem that we know how to solve.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i \in \mathcal{P}} \left\| \mathcal{G}(x_i, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \dots;\, \theta) \right\|_2^2$$

Since the parameters are shared, solving this also gives us the solution network.

$$u(x) \approx \mathcal{N}(x;\, \theta^*)$$

So how do we learn equations?

Consider the following general nonlinear PDE:

$$\mathcal{G}(x, u, \nabla u, \nabla^2 u, \ldots) = 0,$$
$$u = u(x), \quad x \in \Omega$$

Plus boundary conditions…

Approximate the solution of the PDE with an ANN (still untrained)

$$\mathcal{N}(x;\, \theta) \mapsto u$$

It turns out, that we can easily differentiate a neural network, and the **derivative is another network which shares the same parameters as the original**. Remember back-propagation!

Replace the PDE with a neural network.

$$\mathcal{G}(x, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \ldots;\, \theta) = 0$$

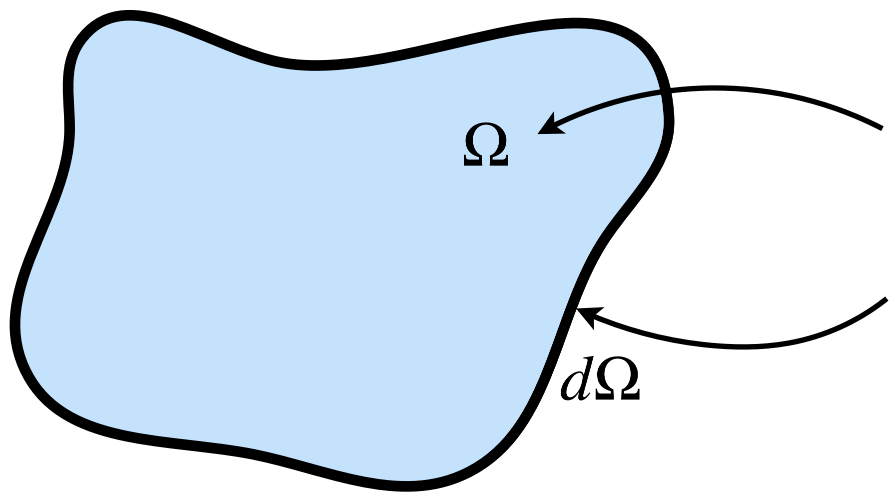
Now we have an optimization problem that we know how to solve.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i \in \mathcal{P}} \left\| \mathcal{G}(x_i, \mathcal{N}, \nabla \mathcal{N}, \nabla^2 \mathcal{N}, \ldots;\, \theta) \right\|_2^2$$

Since the parameters are shared, solving this also gives us the solution network.

$$u(x) \approx \mathcal{N}(x;\, \theta^*)$$

Boundary Conditions

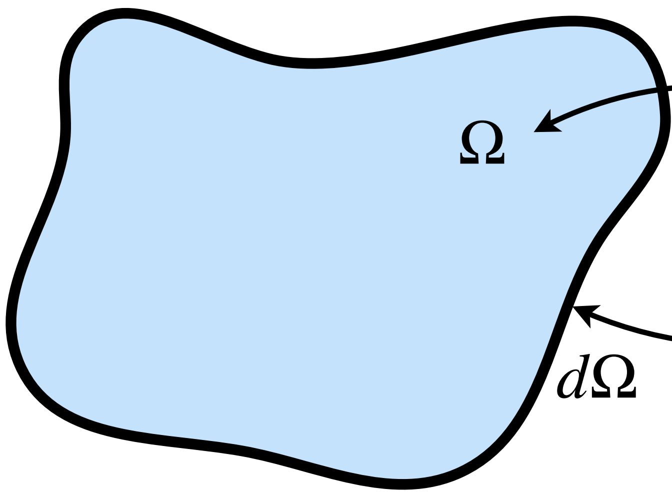


$$\mathfrak{F}[u(x)] = 0, \quad x \in \Omega$$

$$\mathfrak{G}[u(x)] = g(x), \quad x \in d\Omega$$

2 approaches in general…

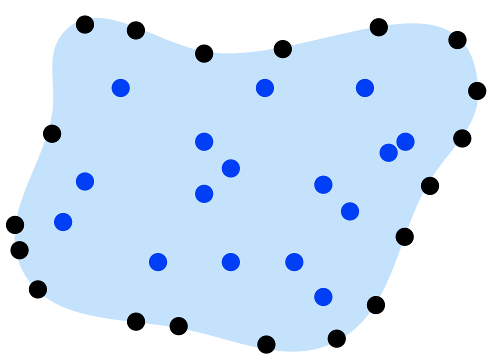
Boundary Conditions



$$\mathfrak{F}[u(x)] = 0, \quad x \in \Omega$$

$$\mathfrak{G}[u(x)] = g(x), \quad x \in d\Omega$$

2 approaches in general…

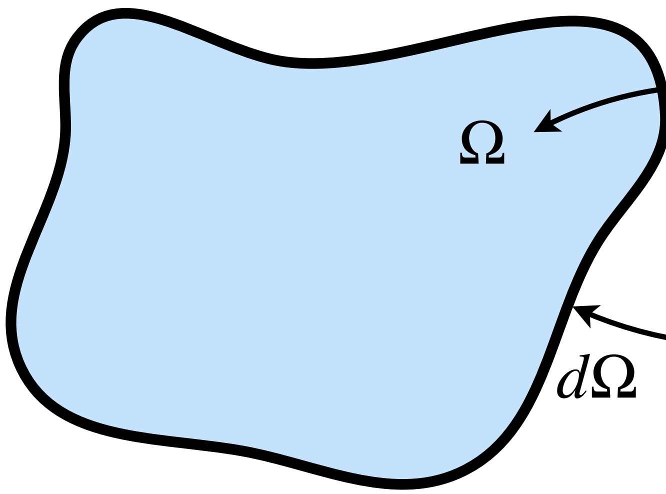


Constrained optimization

$$\hat{u}(x) = \mathcal{N}(x; \theta)$$

$$\mathcal{L}(\theta) = \sum_{x_i \in \Omega} \|\mathfrak{F}[\mathcal{N}(x_i; \theta)]\|_2^2 + \sum_{x_j \in d\Omega} \|\mathfrak{G}[\mathcal{N}(x_j; \theta)] - g(x_j)\|_2^2$$

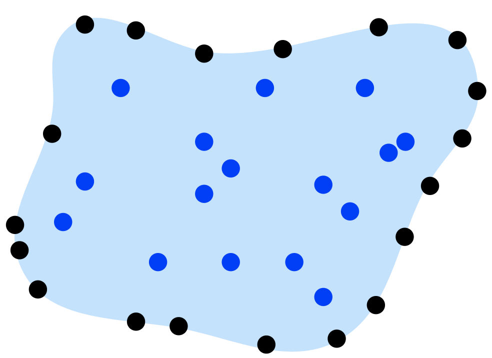
Boundary Conditions



$$\mathfrak{F}[u(x)] = 0, \quad x \in \Omega$$

$$\mathfrak{G}[u(x)] = g(x), \quad x \in d\Omega$$

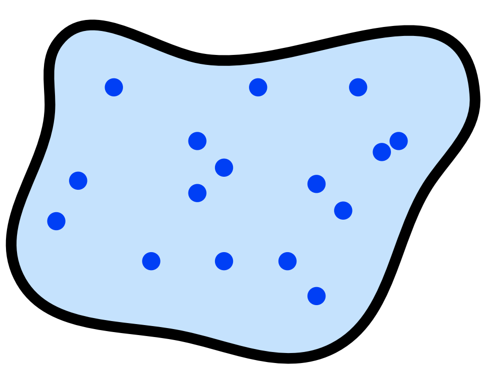
2 approaches in general…



Constrained optimization

$$\hat{u}(x) = \mathcal{N}(x;\theta)$$

$$\mathcal{L}(\theta) = \sum_{x_i \in \Omega} \|\mathfrak{F}[\mathcal{N}(x_i;\theta)]\|_2^2 + \sum_{x_j \in d\Omega} \|\mathfrak{G}[\mathcal{N}(x_j;\theta)] - g(x_j)\|_2^2$$



Unconstrained optimization

$$\hat{u}(x) = A(x) + B(x)\mathcal{N}(x;\theta), \quad \mathfrak{G}[A(x)] = g(x),\ B(x) = 0,\ x \in d\Omega$$

$$\mathcal{L}(\theta) = \sum_{x_i \in \Omega} \|\mathfrak{F}[A(x) + B(x)\mathcal{N}(x;\theta)]\|_2^2$$

Discrete Time Methods

Discrete Time Methods

Consider an unsteady PDE of the form

$$\partial_t u + \mathfrak{F}[u] = 0, \quad (t, x) \in [0,T] \times \Omega \in \mathbb{R}^d$$
$$u(0,x) = g(x), \quad x \in \Omega,$$
$$u(t, x) = h(x), \quad (t, x) \in [0,T] \times d\Omega$$

Discrete Time Methods

Consider an unsteady PDE of the form

$$\partial_t u + \mathfrak{F}[u] = 0, \quad (t, x) \in [0,T] \times \Omega \in \mathbb{R}^d$$
$$u(0,x) = g(x), \quad x \in \Omega,$$
$$u(t, x) = h(x), \quad (t, x) \in [0,T] \times d\Omega$$

General formula for Runge-Kutta time integration

$$u^{n+c_i}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} a_{ij} \mathfrak{F}[u^{n+c_j}(x)]$$

$$u^{n+1}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} b_j \mathfrak{F}[u^{n+c_j}(x)]$$

Discrete Time Methods

Consider an unsteady PDE of the form

$$\partial_t u + \mathfrak{F}[u] = 0, \quad (t, x) \in [0,T] \times \Omega \in \mathbb{R}^d$$
$$u(0,x) = g(x), \quad x \in \Omega,$$
$$u(t, x) = h(x), \quad (t, x) \in [0,T] \times d\Omega$$

General formula for Runge-Kutta time integration

$$u^{n+c_i}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} a_{ij} \mathfrak{F}[u^{n+c_j}(x)]$$
$$u^{n+1}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} b_j \mathfrak{F}[u^{n+c_j}(x)]$$

Put a neural network prior on discrete solutions

$$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)] = \mathcal{N}(x; \theta)$$

Discrete Time Methods

Consider an unsteady PDE of the form

$$\partial_t u + \mathfrak{F}[u] = 0, \quad (t,x) \in [0,T] \times \Omega \in \mathbb{R}^d$$
$$u(0,x) = g(x), \quad x \in \Omega,$$
$$u(t,x) = h(x), \quad (t,x) \in [0,T] \times d\Omega$$

General formula for Runge-Kutta time integration

$$u^{n+c_i}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} a_{ij} \mathfrak{F}[u^{n+c_j}(x)]$$

$$u^{n+1}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} b_j \mathfrak{F}[u^{n+c_j}(x)]$$

Put a neural network prior on discrete solutions

$$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)] = \mathcal{N}(x;\theta)$$

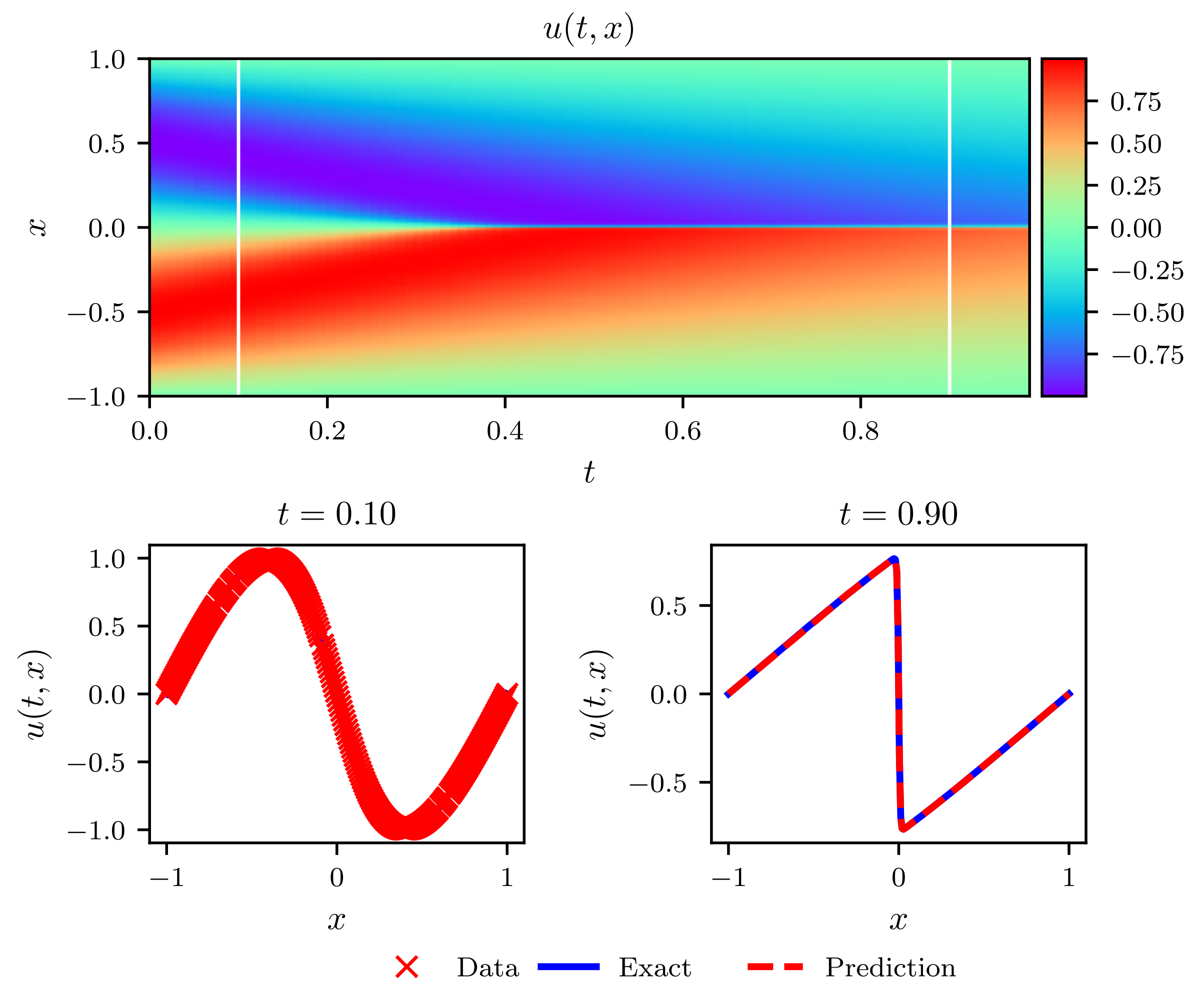
Inserting network into RK scheme yields desired minimization problem based on known solution at time level n

Enables very high-order schemes!

Discrete Time Methods

Consider an unsteady PDE of the form

$$\partial_t u + \mathfrak{F}[u] = 0, \quad (t, x) \in [0,T] \times \Omega \in \mathbb{R}^d$$
$$u(0,x) = g(x), \quad x \in \Omega,$$
$$u(t,x) = h(x), \quad (t,x) \in [0,T] \times d\Omega$$



General formula for Runge-Kutta time integration

$$u^{n+c_i}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} a_{ij} \mathfrak{F}[u^{n+c_j}(x)]$$

$$u^{n+1}(x) = u^n(x) - \Delta t \sum_{j=1}^{q} b_j \mathfrak{F}[u^{n+c_j}(x)]$$

Put a neural network prior on discrete solutions

$$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)] = \mathcal{N}(x; \theta)$$

$O(\Delta t^{1000})$

Inserting network into RK scheme yields desired minimization problem based on known solution at time level n

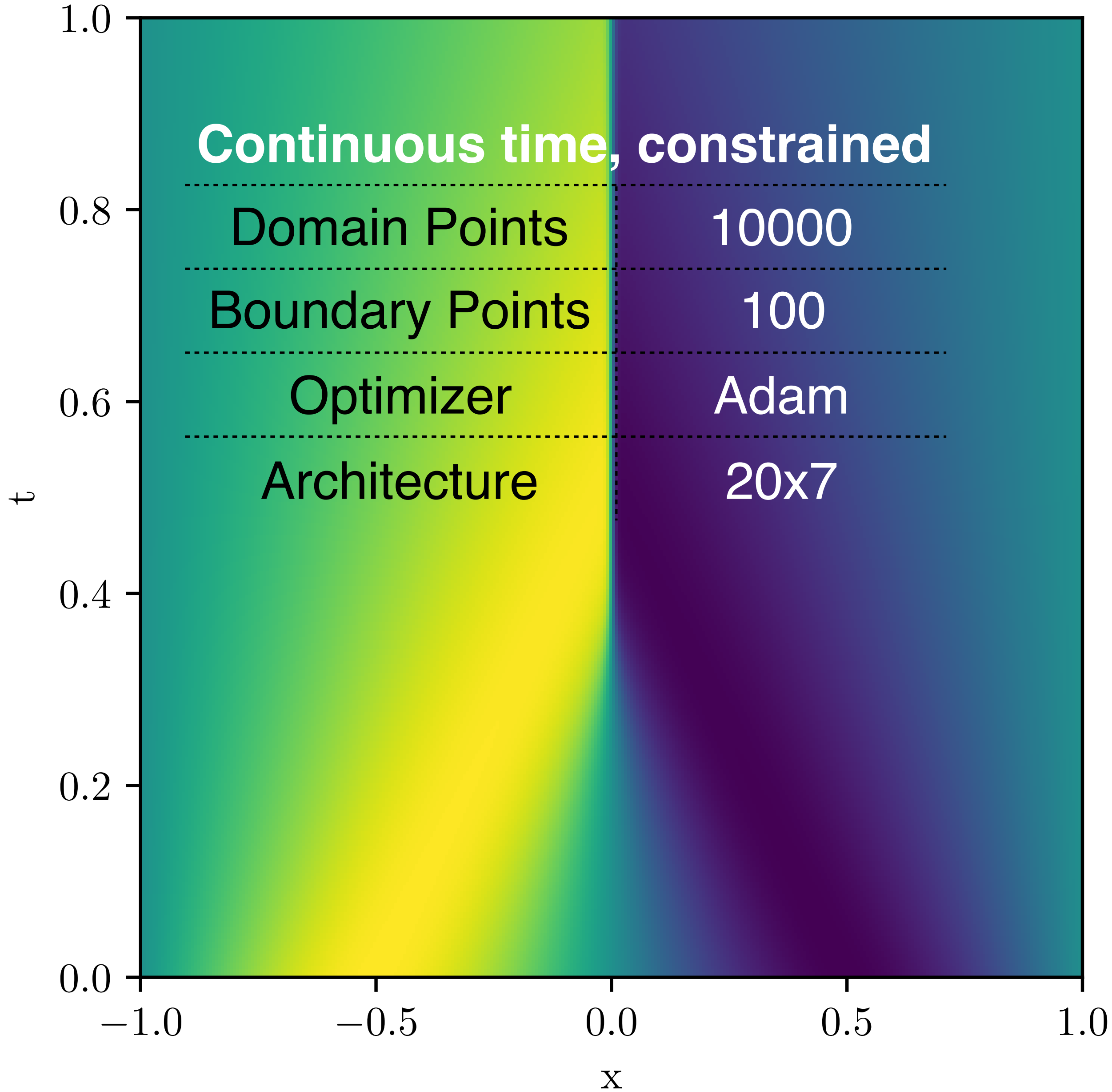
Enables very high-order schemes!

M. Raissi et al. arXiv:1711.10561v1, 2017.

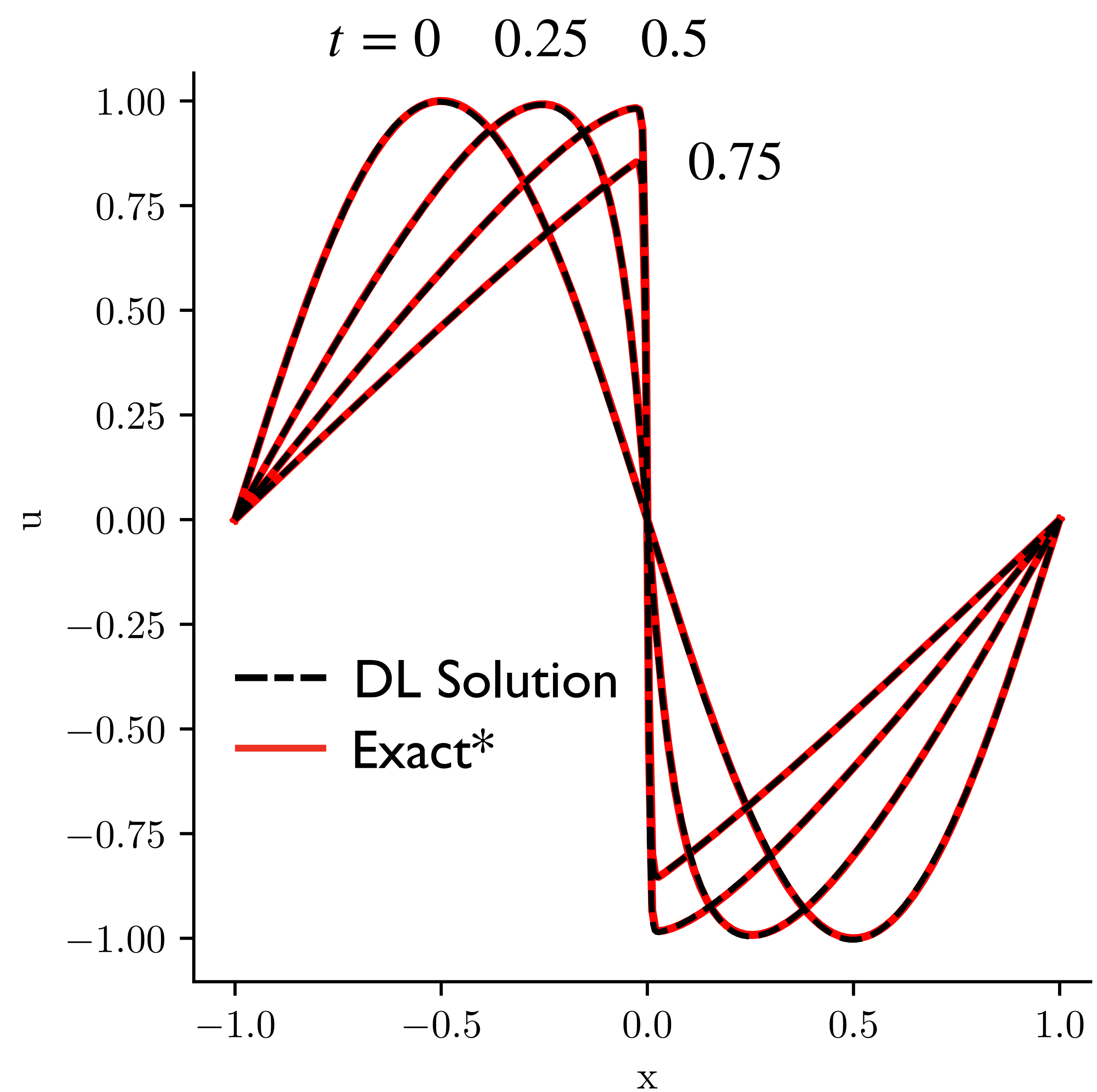
Proven on simple problems

Burgers equation with smooth opposing waves

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1,1], \quad t \in [0,1],$$
$$u(0,x) = -\sin(\pi x),$$
$$u(t,-1) = u(t,1) = 0.$$



Continuous time, constrained	
Domain Points	10000
Boundary Points	100
Optimizer	Adam
Architecture	20x7



*M. Raissi et al. arXiv:1711.10561v1, 2017.

Probing for weakness on hyperbolic systems

Entropic solution of inviscid Burgers equation

$$\partial_t u + \frac{1}{2}\partial_x u^2 = \nu \partial_{xx} u, \quad u = u(t,x), \quad t,x \in \mathbb{R}_+ \times \mathbb{R}, \quad \nu \to 0$$

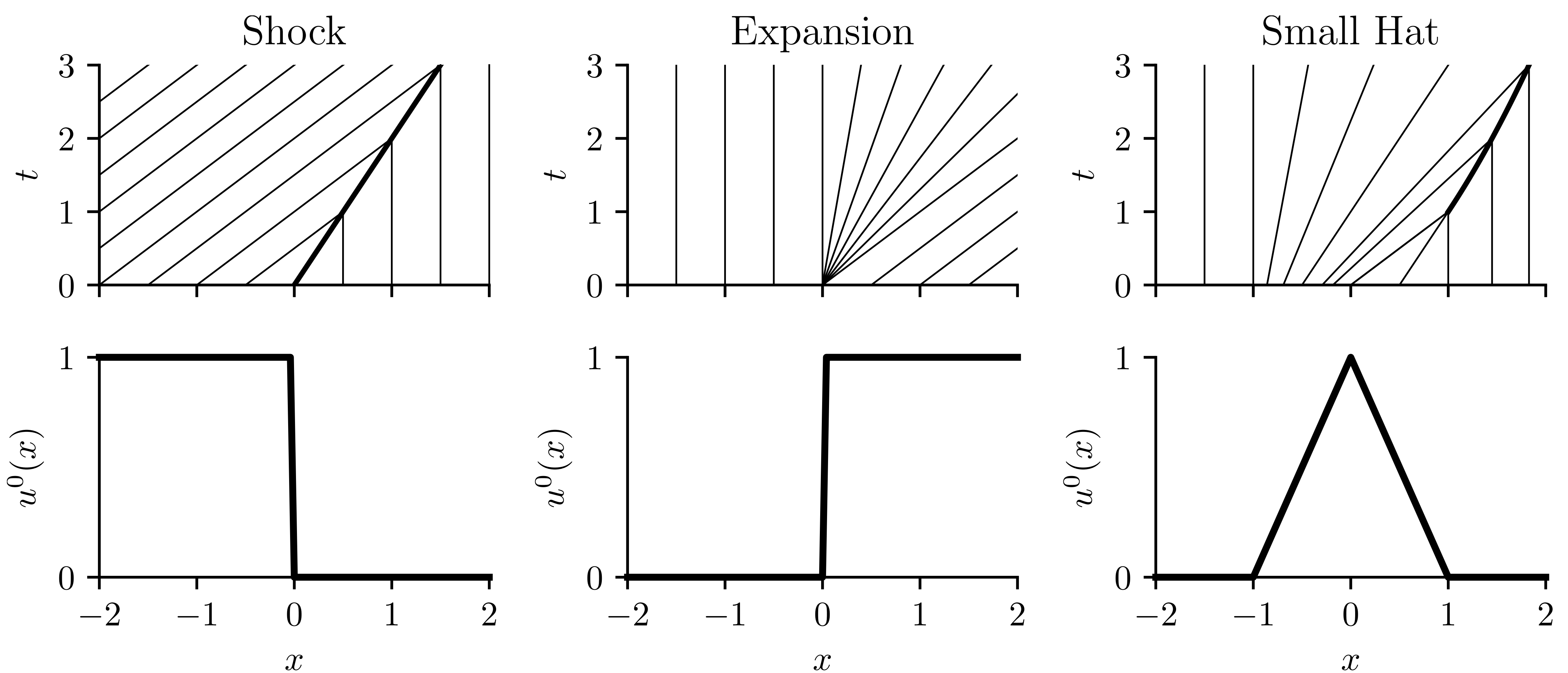
$$u(0,x) = u^0(x)$$

Probing for weakness on hyperbolic systems

Entropic solution of inviscid Burgers equation

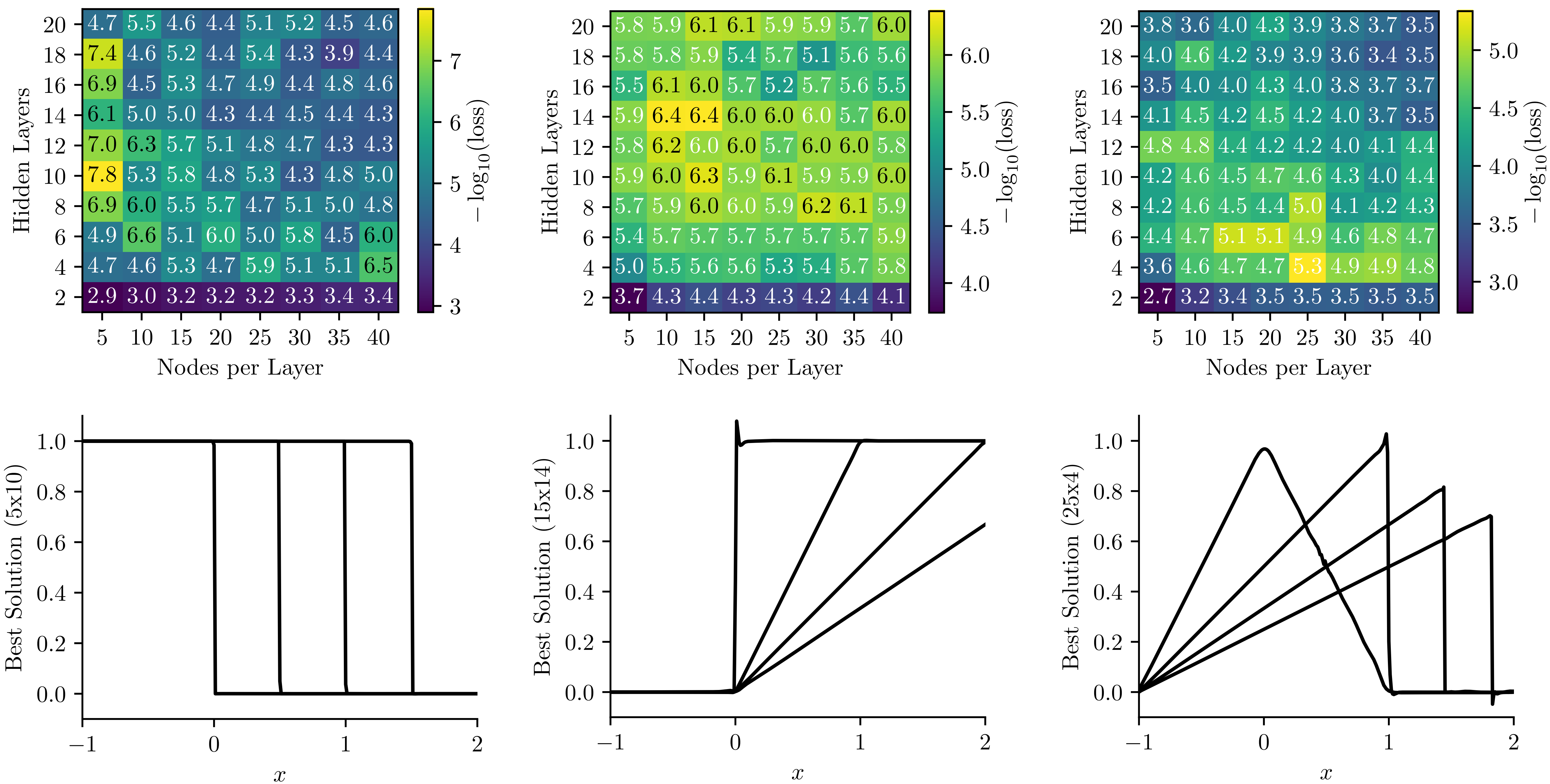
$$\partial_t u + \frac{1}{2}\partial_x u^2 = \nu \partial_{xx} u, \quad u = u(t,x), \quad t,x \in \mathbb{R}_+ \times \mathbb{R}, \quad \nu \to 0$$

$$u(0,x) = u^0(x)$$

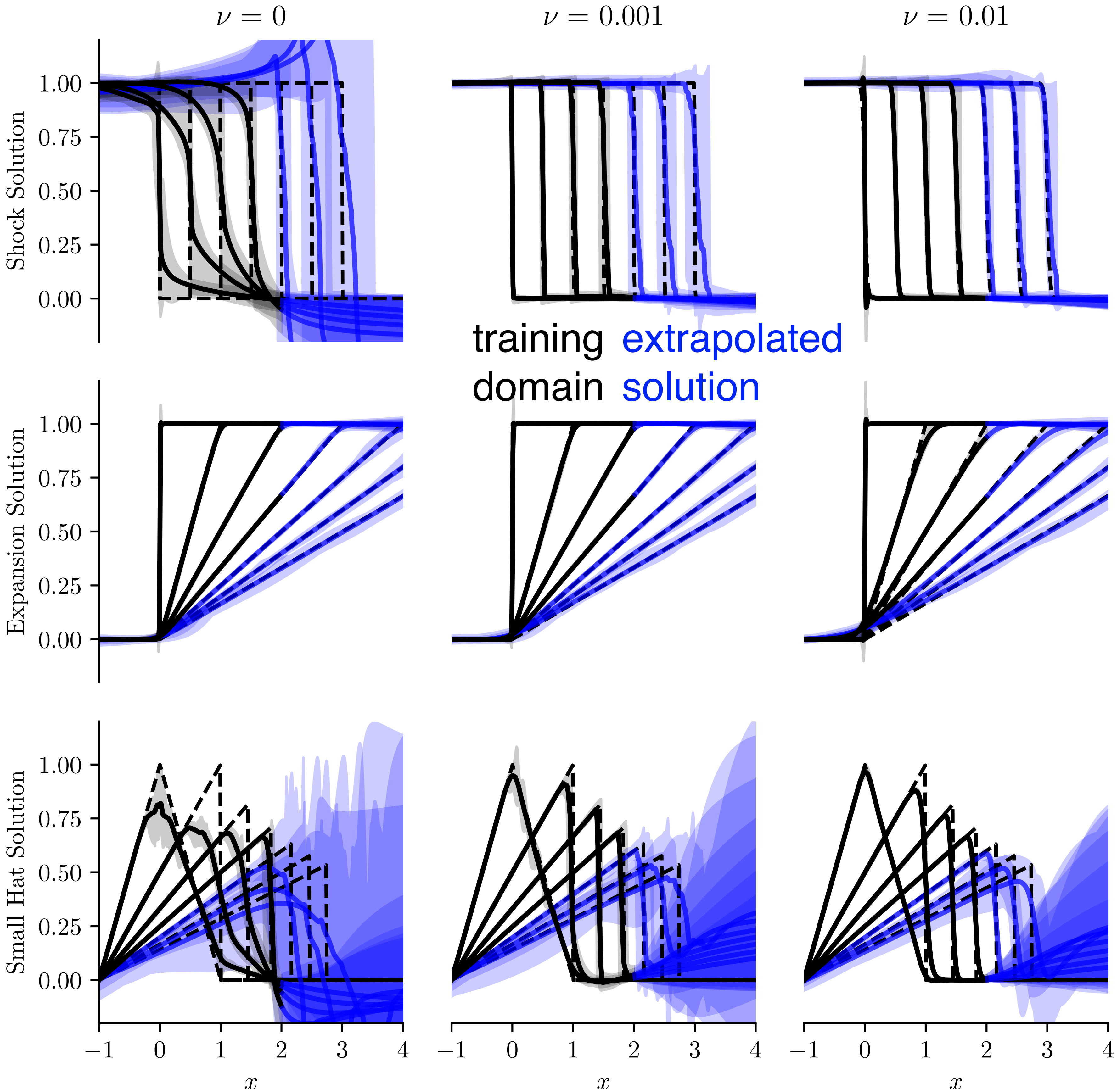


Representation of solutions with ANNs

Parametric regression study with dense, feed-forward networks



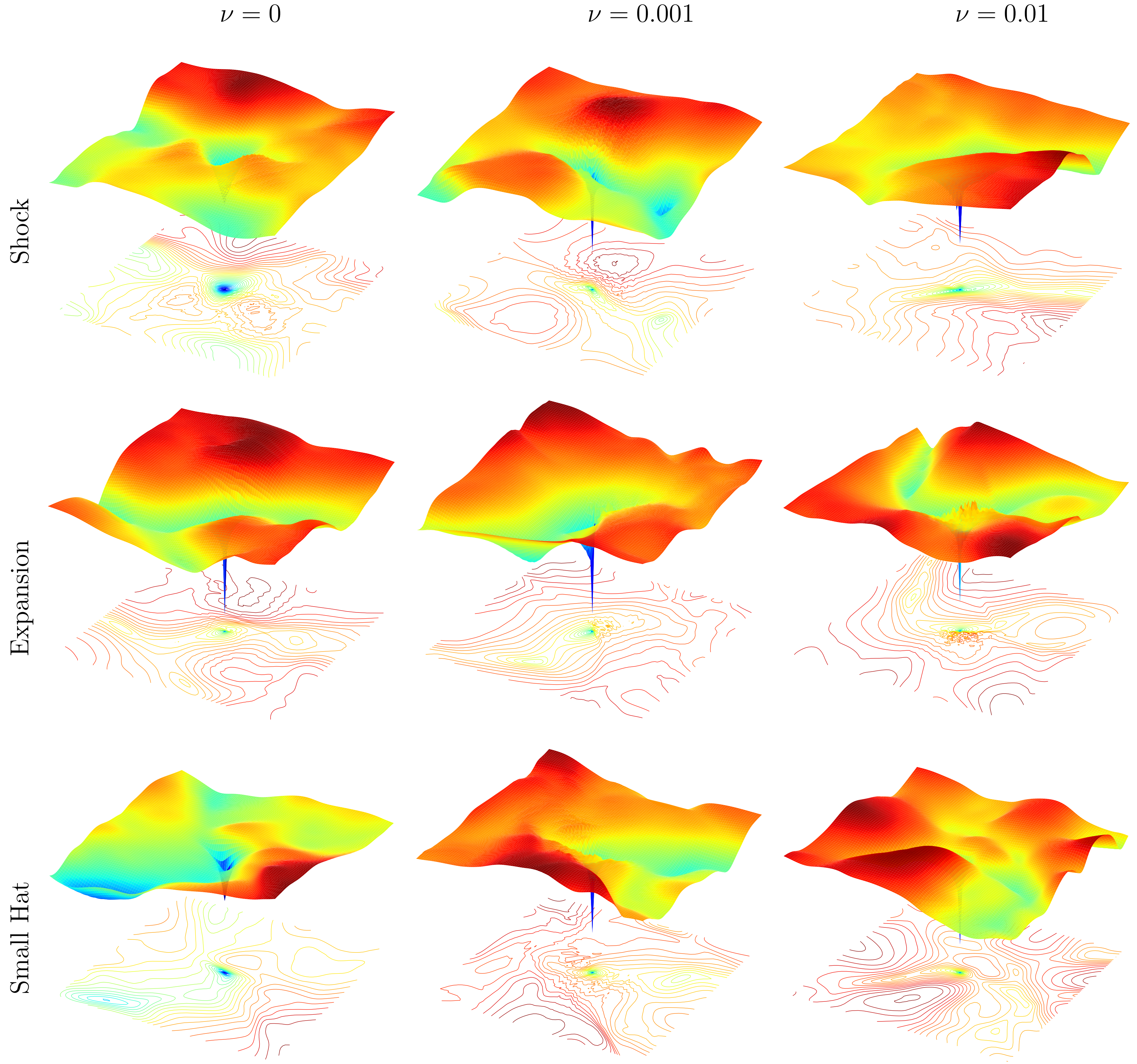
Solutions with 7 hidden layers of 20 nodes



- 25 unique solutions
- 3 viscosities
- Solution envelopes

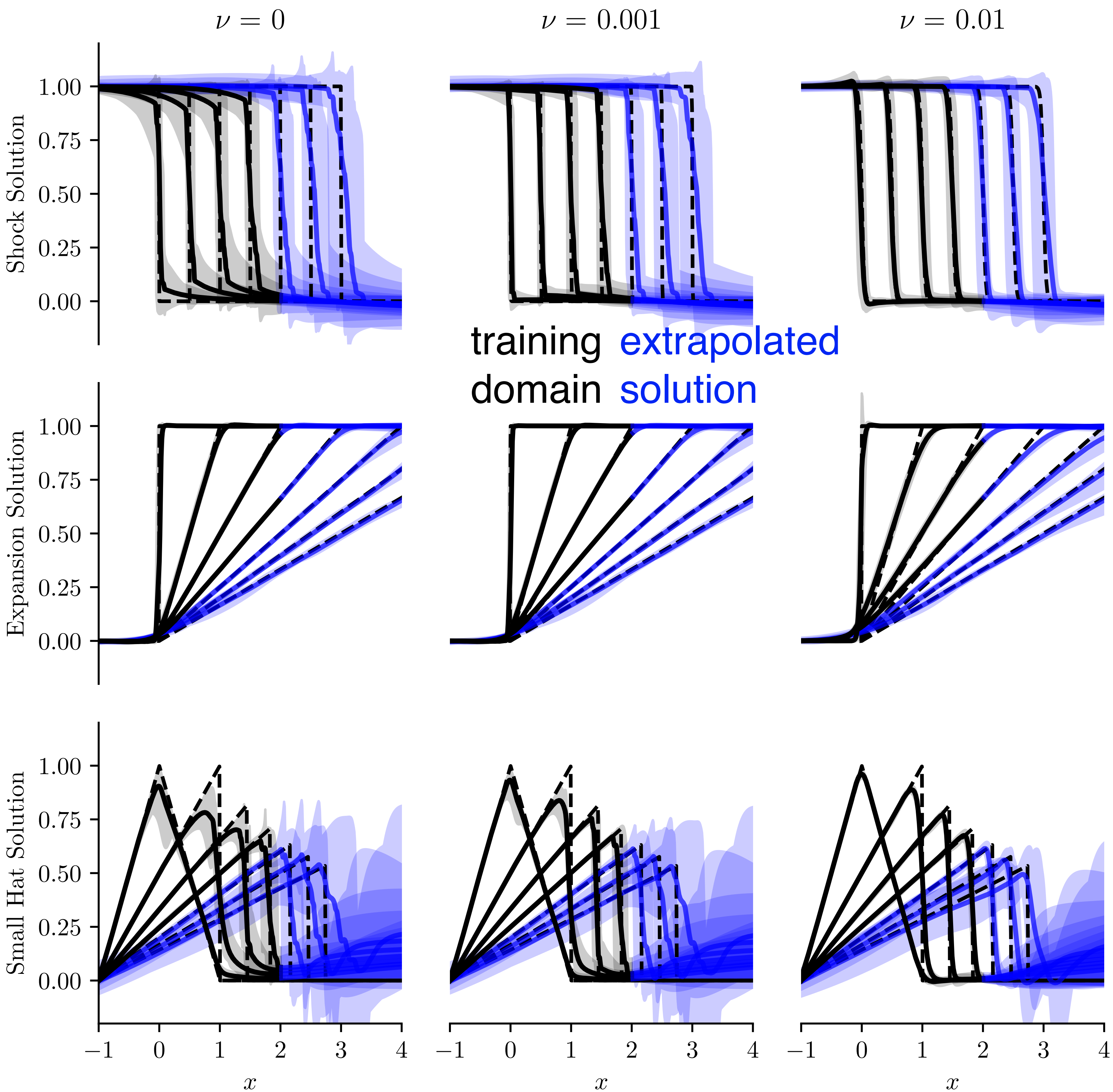
- Good generalization outside of training domain
- More accurate/ certain solution with increasing viscosity

Projected loss surfaces provide a clue



Li, Xu, Taylor, Studer, Goldstein. *arXiv:1712.09913 [cs.LG]*, 2018.

Treating viscosity as another dimension



- Better generalization for low viscosity
- Smaller variance
- Closer to entropic solution for inviscid case
- Possible that network expressibility reached

Concluding Remarks

Introduction to deep learning techniques for solving PDEs

- ANNs may help us overcome issues related to classical discretization schemes
- Break free from the curse of dimensionality
- Deep NNs have proven to be very successful at representing complex functions
- Inserting a NN in the PDE and BCs with colocation yields optimization problem
- Variety of ways to treat boundary conditions, time integration, sampling, …

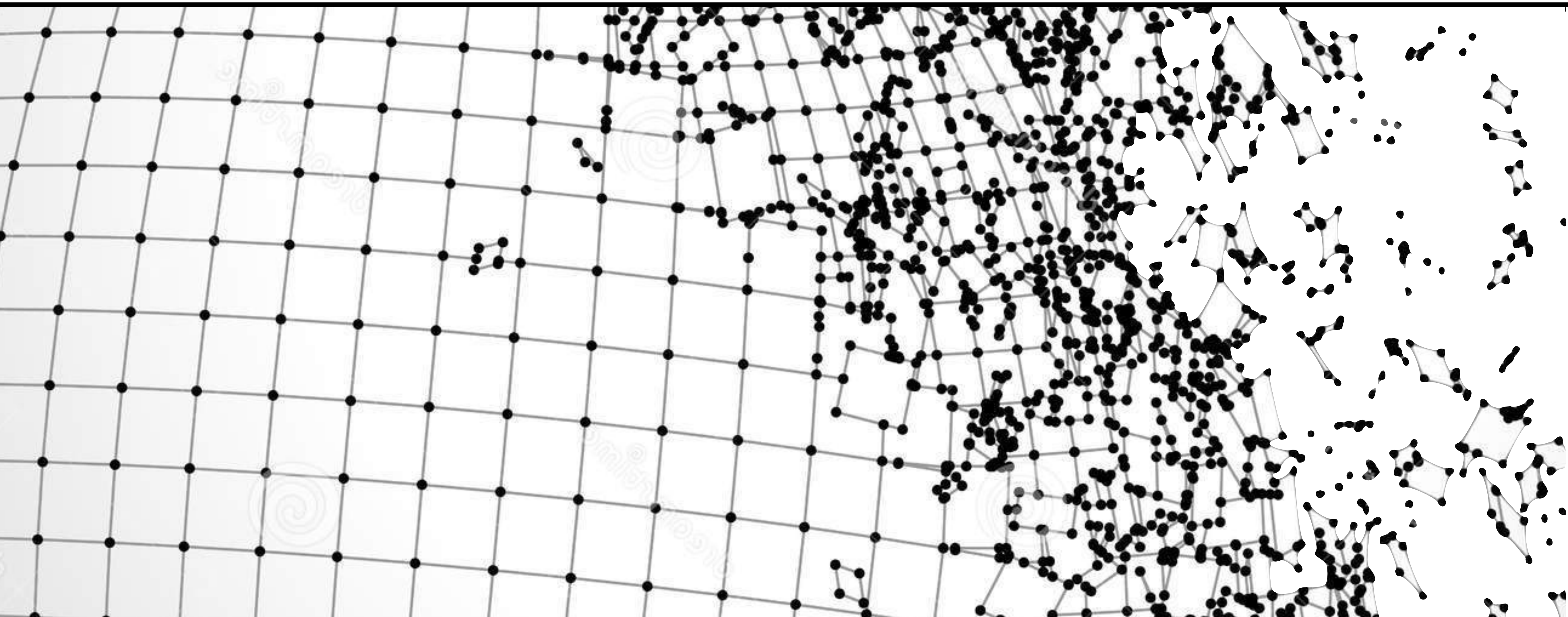
Irregular/discontinuous solutions are difficult to train with current techniques

- Viscous Burgers equation is easier to solve with increasing viscosity (dissipation)
- Inviscid solutions have more variance and lower accuracy
- Generalizing the solution on a range of viscosities seems to improve the situation

Promising, but there is a lot of work left to be done!

- Next talks look at the approximation capacity of DNNs as well as an alternative method based on LS-SVM, stick around!

Solving Partial Differential Equations with Deep Learning



James B. Scoggins
www.jbscoggins.com
@jb_scoggins