SINTEF

GitHub repo with exercises:
tinyurl.com/pimlws

Structure-preserving machine learning for physical systems

Sølve Eidnes, SINTEF Digital

NLDL winter school, January 6 2025

Norwegian AI Cloud

PhysML

Outline

1st hour:

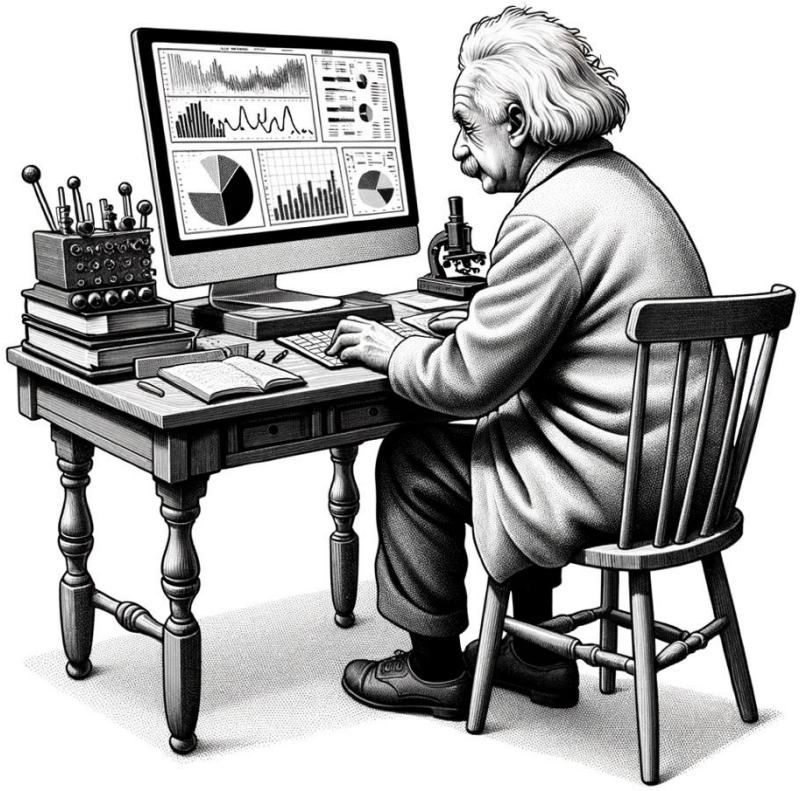
- Introduction to physics-informed machine learning
  - Why do physics-informed ML?
  - Soft vs hard constraints
- Hamiltonian mechanics and the mass-spring system
- Physics-informed neural networks (PINN)
  - Exercise 0: PINN for damped mass-spring system

2nd hour:

- Hamiltonian neural networks (HNN)
- Pseudo-Hamiltonian neural networks (PHNN) for ODEs
  - Exercise 1: PHNN for damped mass-spring system

3rd hour:

- Pseudo-Hamiltonian neural networks for PDEs
  - Exercise 2: HNN for the KdV equation



Ill.: ChatGPT/Dall-E 3

Part 1:

Introduction to physics-informed machine learning

SINTEF

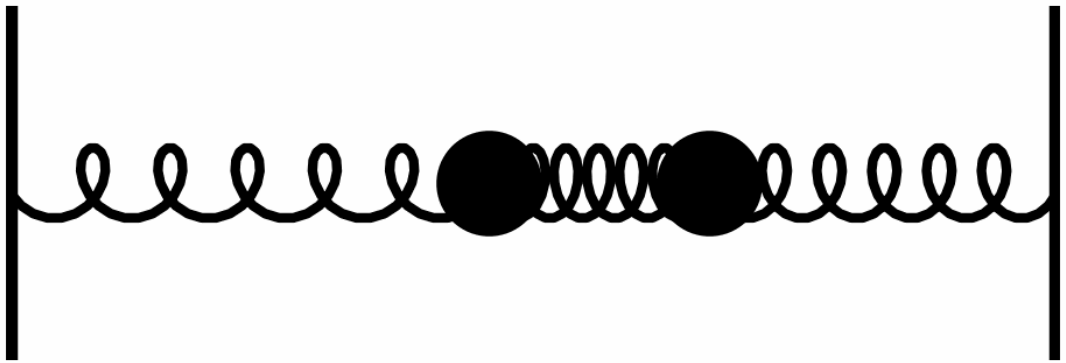
$$\dot{q}_i = \frac{\partial H}{\partial p_i}$$

$$\dot{p}_i = -\frac{\partial H}{\partial q_i}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0$$

$$\frac{\partial S}{\partial t} + H\left(q_i, \frac{\partial S}{\partial q_i}, t\right) = 0$$

$$\frac{d\rho}{dt} = \{\rho, H\} = 0$$

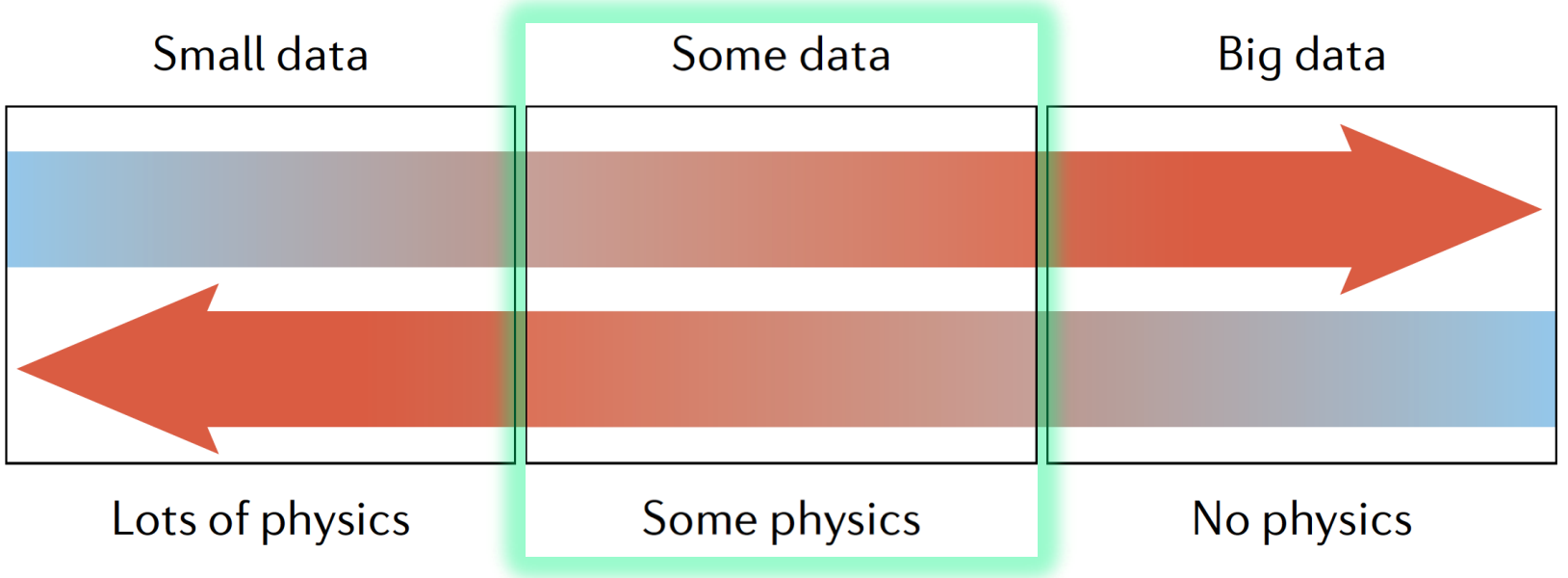


$$\omega = \sum_{i=1}^{n} dq_i \wedge dp_i$$

$$\delta \int_{t_1}^{t_2} L\, dt = 0$$

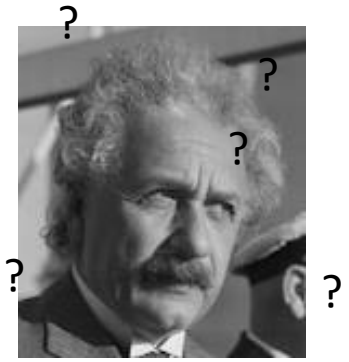
$$\dot{x} = (S(x) - R(x))\nabla H(x) + f(x,t)$$

Why and when inform ML models about physics?



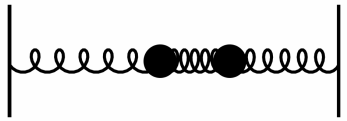
Karniadakis et al. "Physics-informed machine learning." *Nature Reviews Physics* 3.6 (2021): 422-440.

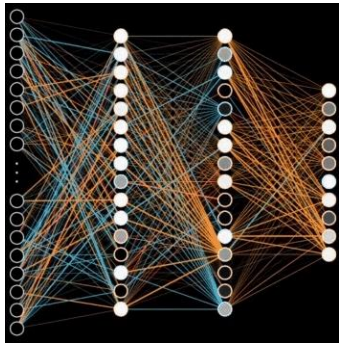
Training process



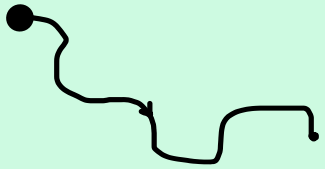
Getty images

Data





Learnable function space



Learned model

Space of all possible models a neural net can learn

- More data?
- Larger model?
- More compute?
- More data scientists?

SINTEF

Training process

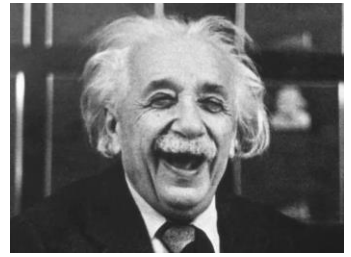
$$\delta \int_{t_1}^{t_2} L\,dt = 0$$

$$\dot{x} = (S(x) - R(x))\nabla H(x) + f(x,t)$$

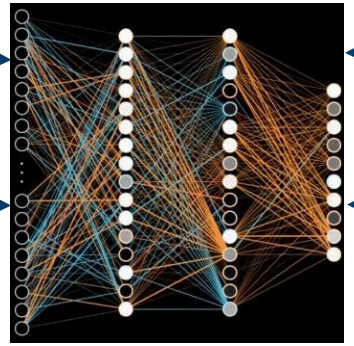
$$\dot{q}_i = \frac{\partial H}{\partial p_i}$$

$$\dot{p}_i = -\frac{\partial H}{\partial q_i}$$

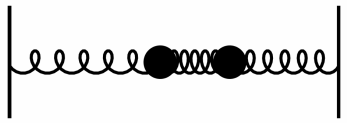
$$\omega = \sum_{i=1}^{n} dq_i \wedge dp_i$$



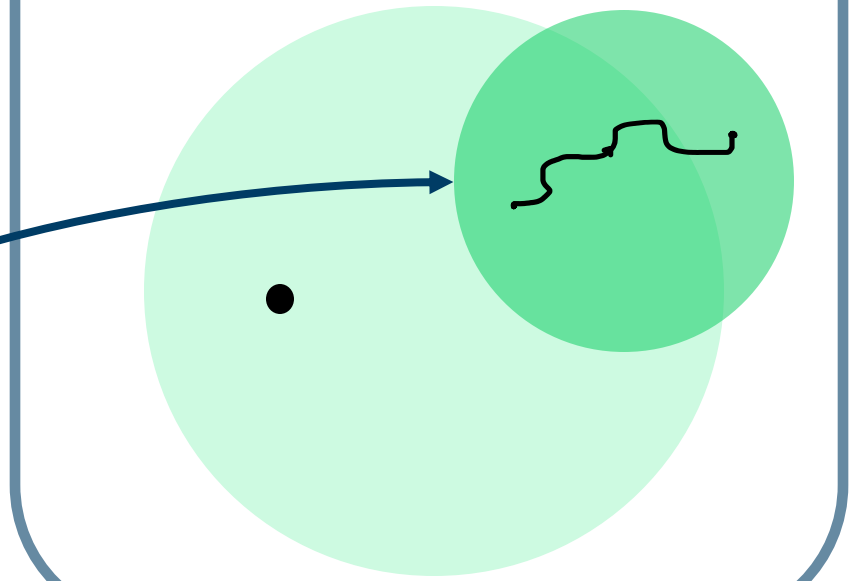
Getty images



Data



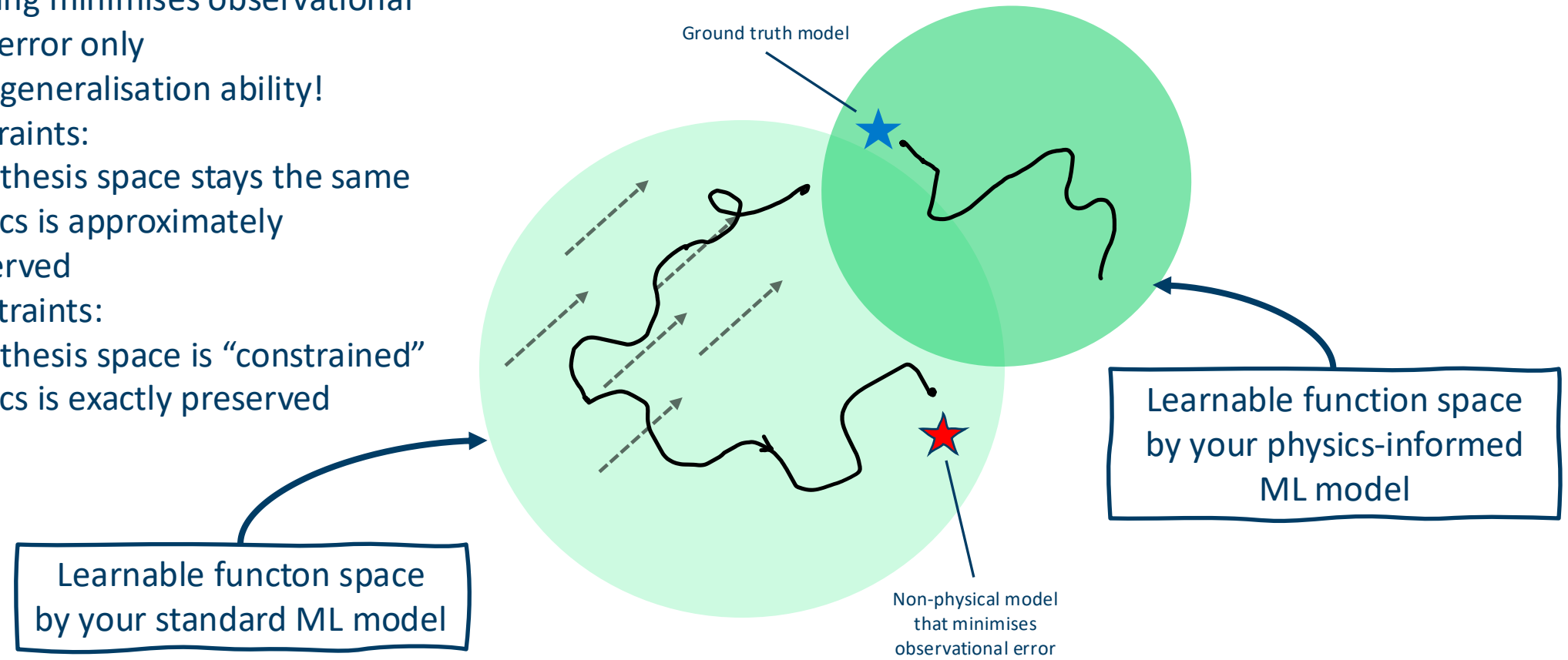
Learned model



Space of all physically possible models

Soft vs. hard constraints

- No constraints:
  - Training minimises observational data error only
  - Poor generalisation ability!
- Soft constraints:
  - Hypothesis space stays the same
  - Physics is approximately preserved
- Hard constraints:
  - Hypothesis space is “constrained”
  - Physics is exactly preserved



Hamiltonian mechanics and our test case: the mass-spring system

Hamiltonian mechanics

Define q and p as generalized position and momentum. A canonical Hamiltonian system is given by

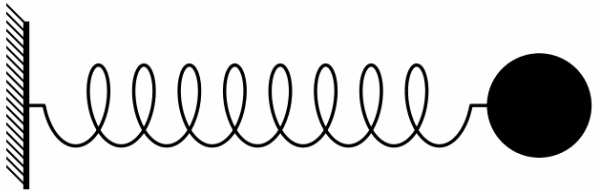
$$\begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{H}}{\partial q} \\ \frac{\partial \mathcal{H}}{\partial p} \end{bmatrix},$$

where the Hamiltonian $\mathcal{H}(q, p)$ represents the total energy.

The Hamiltonian is an invariant, preserved along the solution curves of the system. That is,

$$\mathcal{H}(q(t), p(t)) = \mathcal{H}(q(t_0), p(t_0)) \quad \text{for all } t.$$

Mass-spring system



Consider a simple mass-spring system, *without* damping or external forces. The force from the spring is $F = -k(x - x_0)$. Together with Newton's law $F = m\ddot{x}$, you get ($x_0 = 0$)

$$m\ddot{x} + kx = 0.$$

With $q = x$ and $p = m\dot{x}$, the total energy is

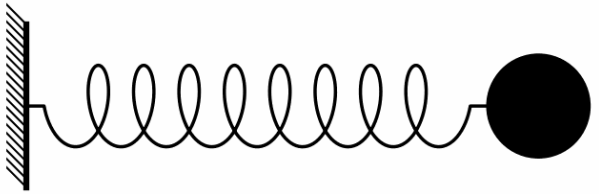
$$\mathcal{H}(q, p) = T(p) + V(q) = \frac{1}{2m}p^2 + \frac{k}{2}q^2,$$

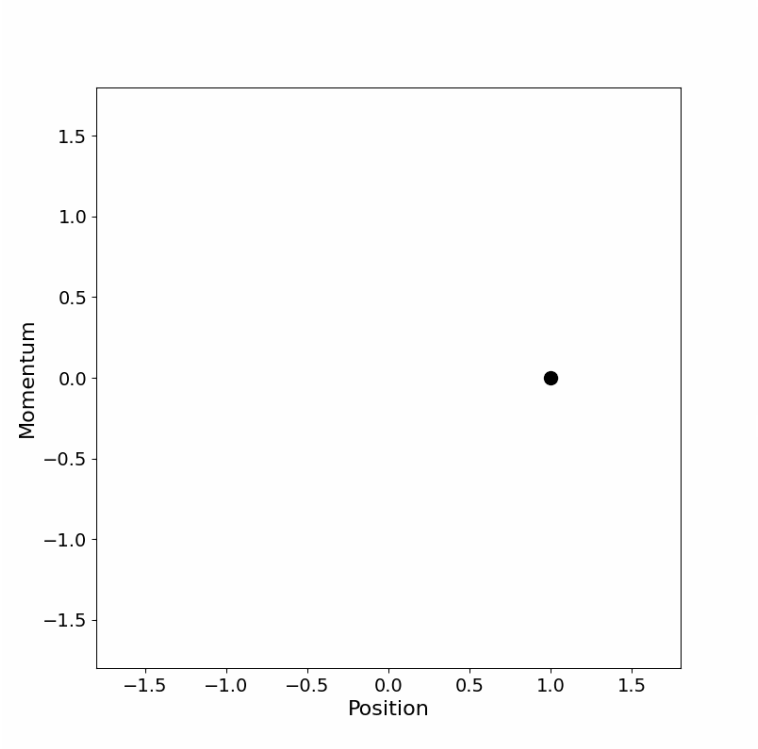
and Hamilton's equations

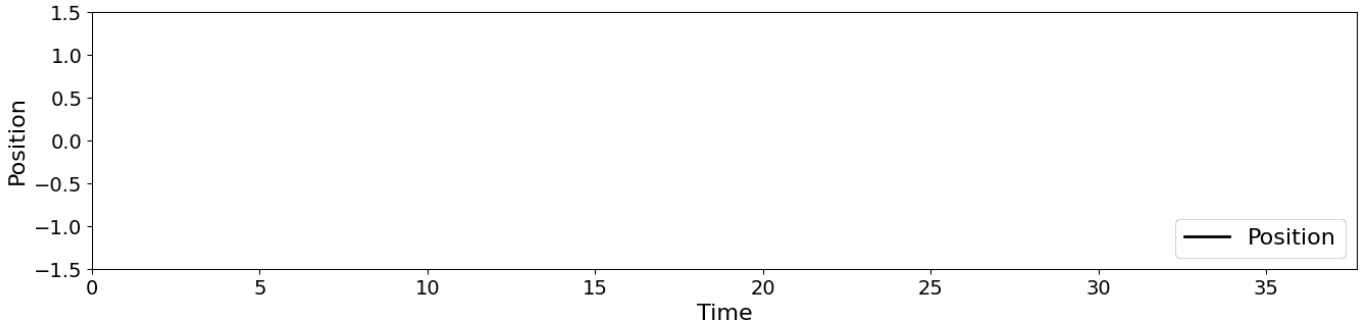
$$\dot{q} = \frac{\partial \mathcal{H}}{\partial p} = \frac{1}{m}p, \qquad \dot{p} = -\frac{\partial \mathcal{H}}{\partial q} = -kq$$

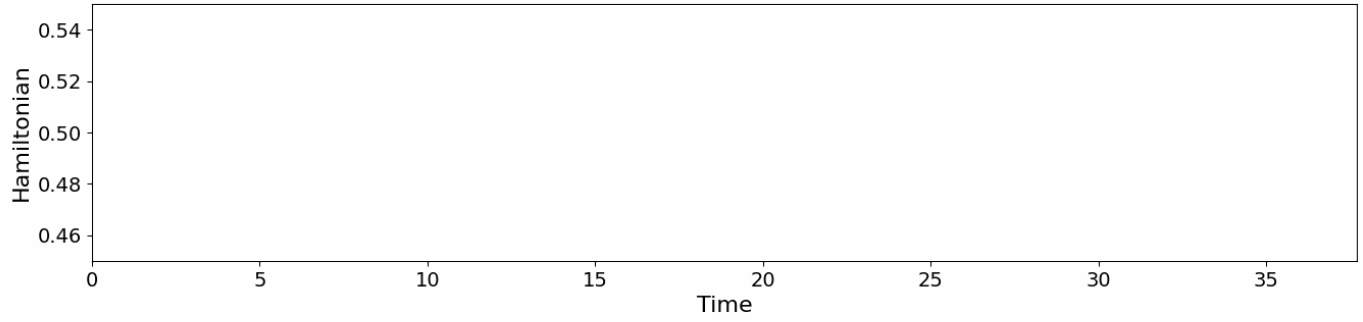
are equivalent to $m\ddot{q} + kq = 0$.

Mass-spring system

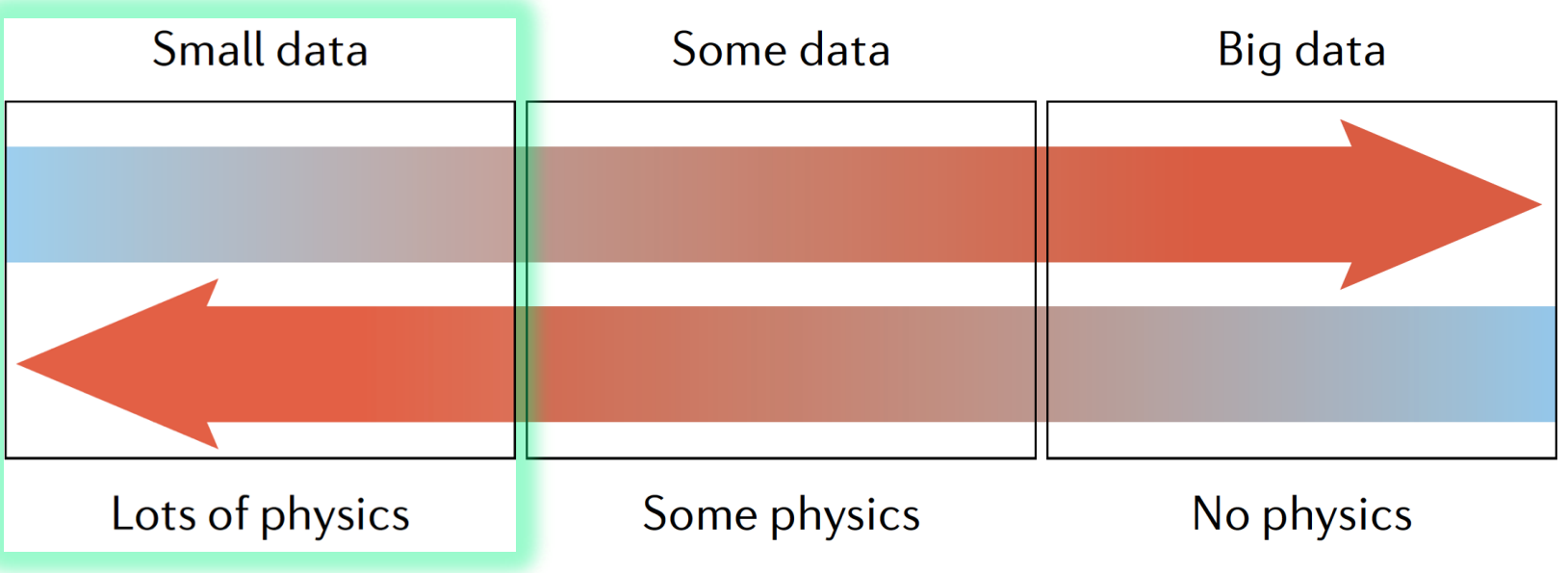






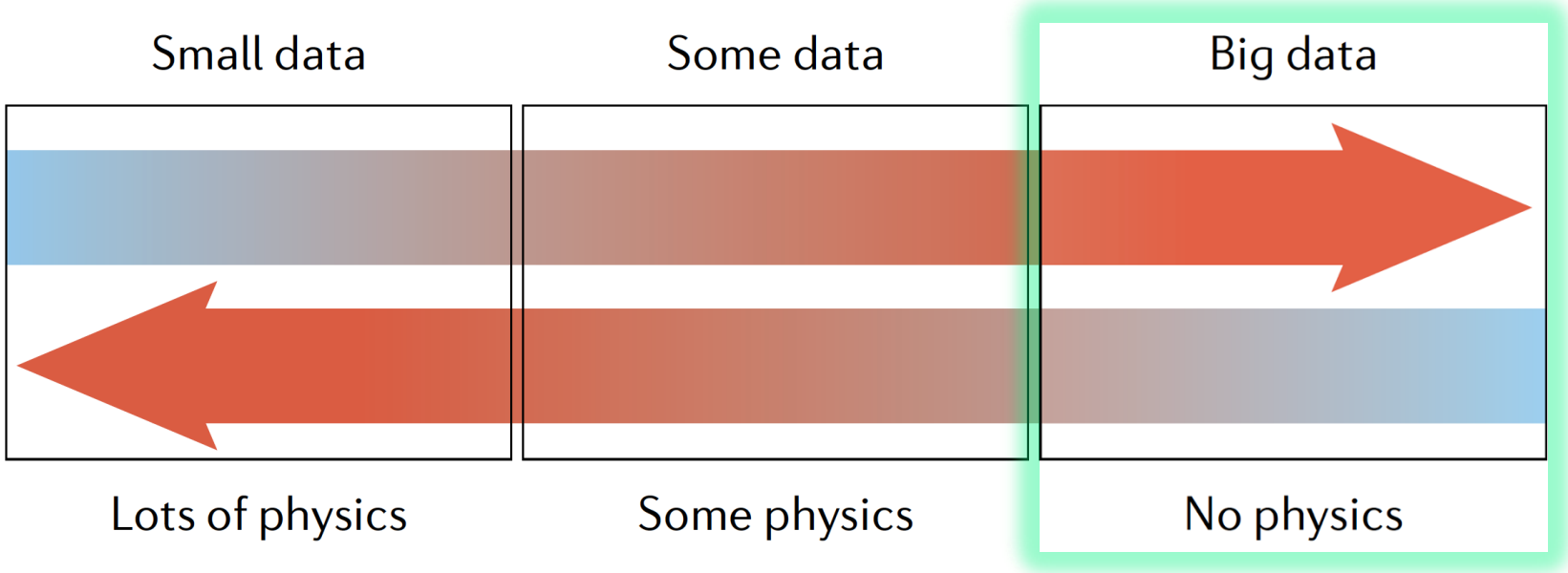


Mass-spring system



Know $m\ddot{x} + kx = 0$, and know m and k
→ Use numerical integration

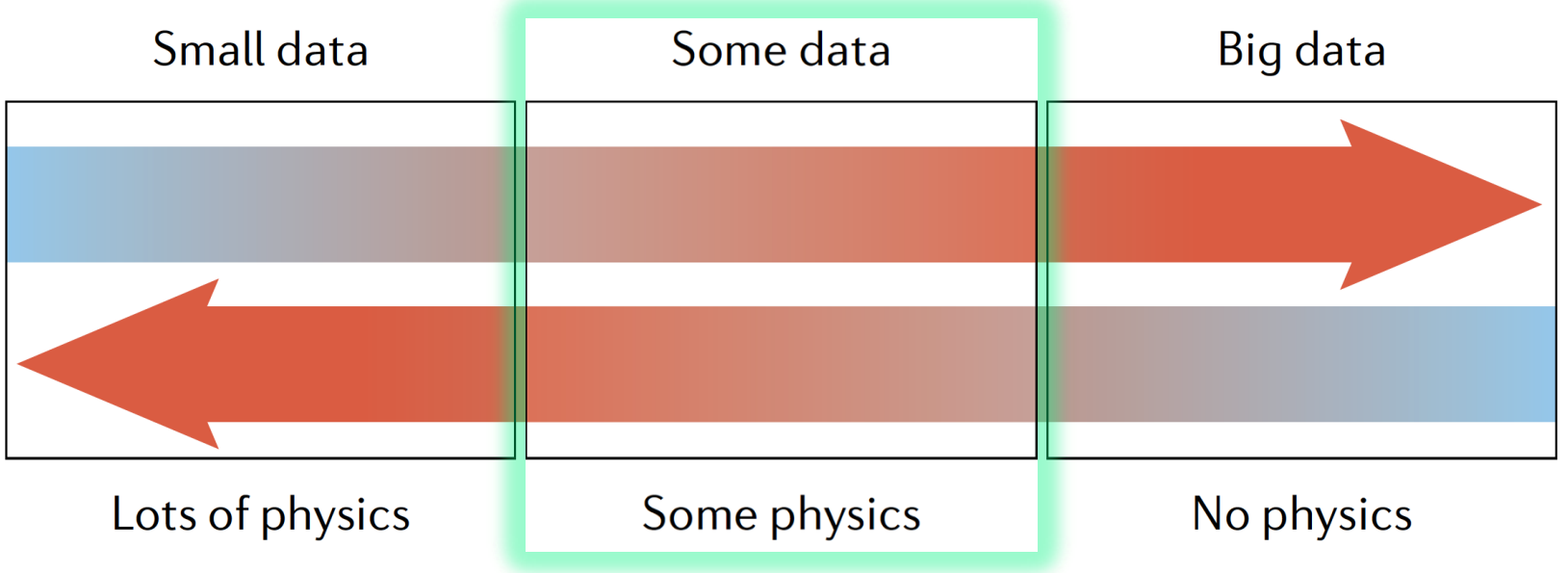
Mass-spring system



Don't know $m\ddot{x} + kx = 0$, or that it's a Hamiltonian system

→ Use purely data-driven learning

Mass-spring system



Know $m\ddot{x} + kx = 0$, but not m and/or k,
or know

$$\begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{H}}{\partial q} \\ \frac{\partial \mathcal{H}}{\partial p} \end{bmatrix}$$

but not $\mathcal{H}$
→ Use physics-informed ML

Part 2:

Physics-informed neural networks

Physics informed neural networks (PINN)

- We assume to know the differential equation $ODE[x, k]$, except for the the physical constant k
- We also have some observational data t_i , x_i
- Here we should be able to use the data to infer what k must be, while inferring the solution x_j at any t_j
- How can we use the data plus the ODE to find $x(t)$?
  - Train a neural network model on the *informed* loss function

$$L = \sum_{t_i,x_i} |x^{\theta}(t_i) - x_i| + \sum_{t_j} |m\ddot{x}^{\theta}(t_j) - k\mathrm{x}^{\theta}(t_j)|$$

PINN Loss = **Traditional, data-driven loss** + **Additional physics loss**

Raissi, Perdikaris, and Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." *Journal of Computational physics* 378 (2019): 686-707.

Physics informed neural networks

In general, for the differential equation $DE[u, \lambda] = 0$ the loss function

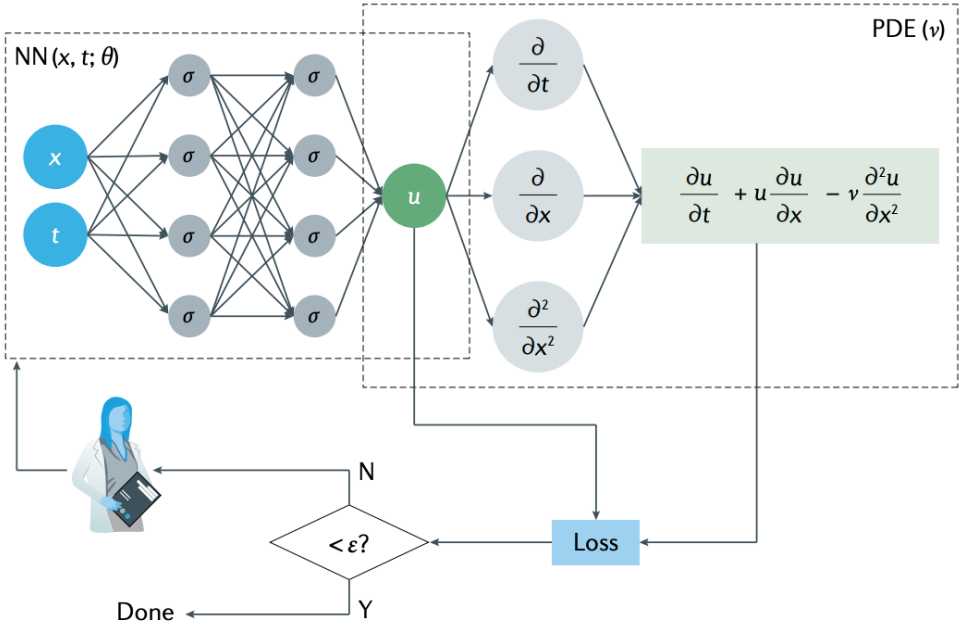
$$L = \sum |u^{\theta}(x_i, t_i) - u_i|^2 + \sum |DE[u^{\theta}(x_i, t_i), \lambda]|^2$$

is (weakly) constraining the neural net to learn a form that satisfies

$$DE[u^{\theta}, \lambda] < tol$$

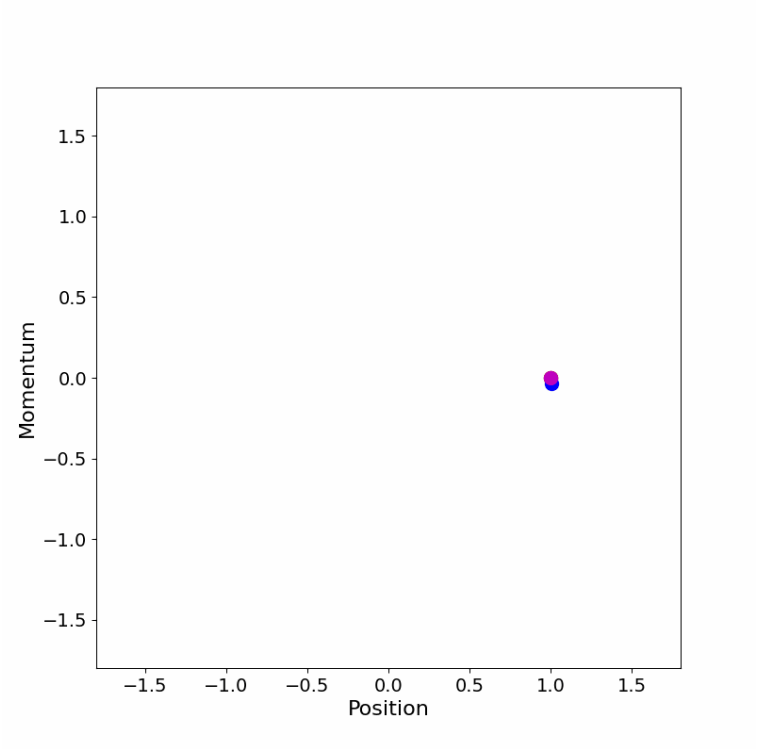
The equation $DE[u^{\theta}, \lambda]$ can contain partial derivatives w.r.t. time and space

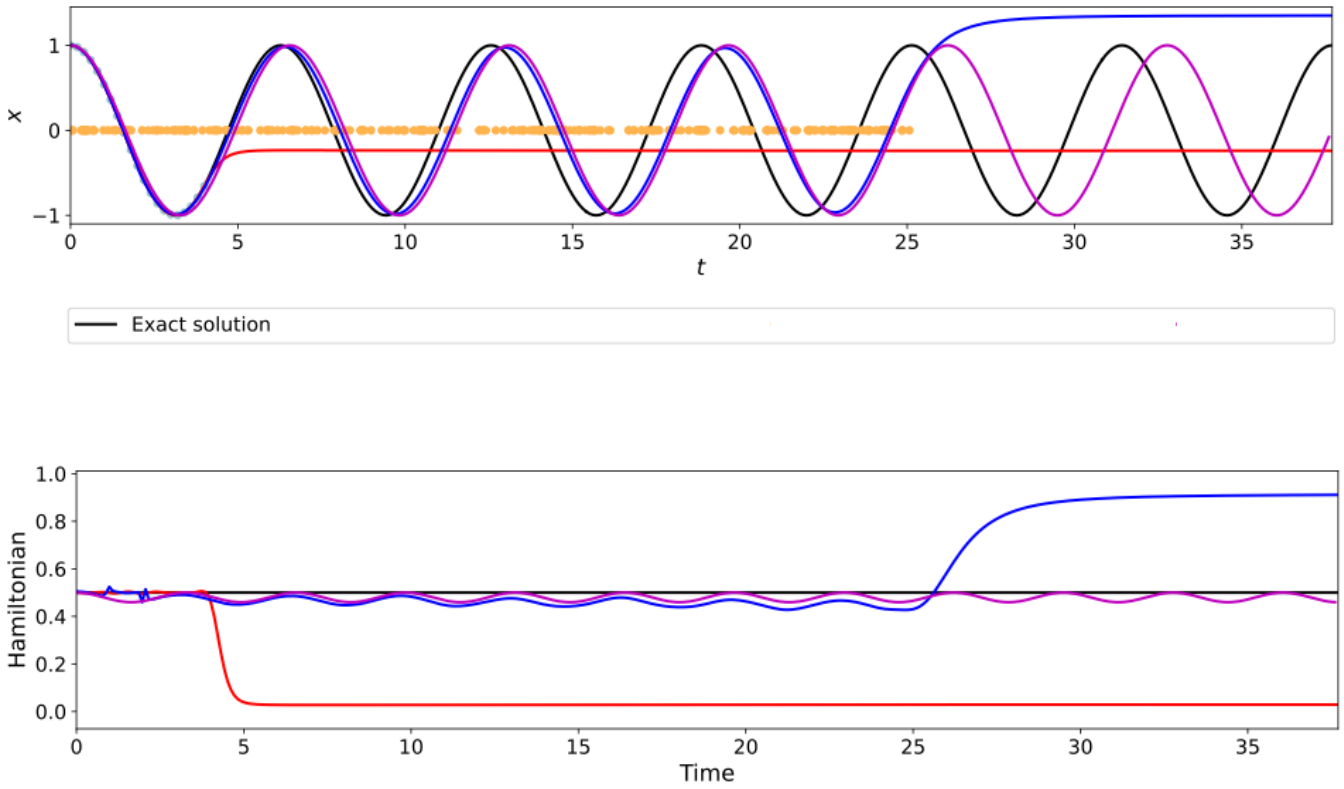
- This is computed using automatic differentiation



Karniadakis, George Em, et al. "Physics-informed machine learning." *Nature Reviews Physics* 3.6 (2021): 422-440

PINN for mass-spring system





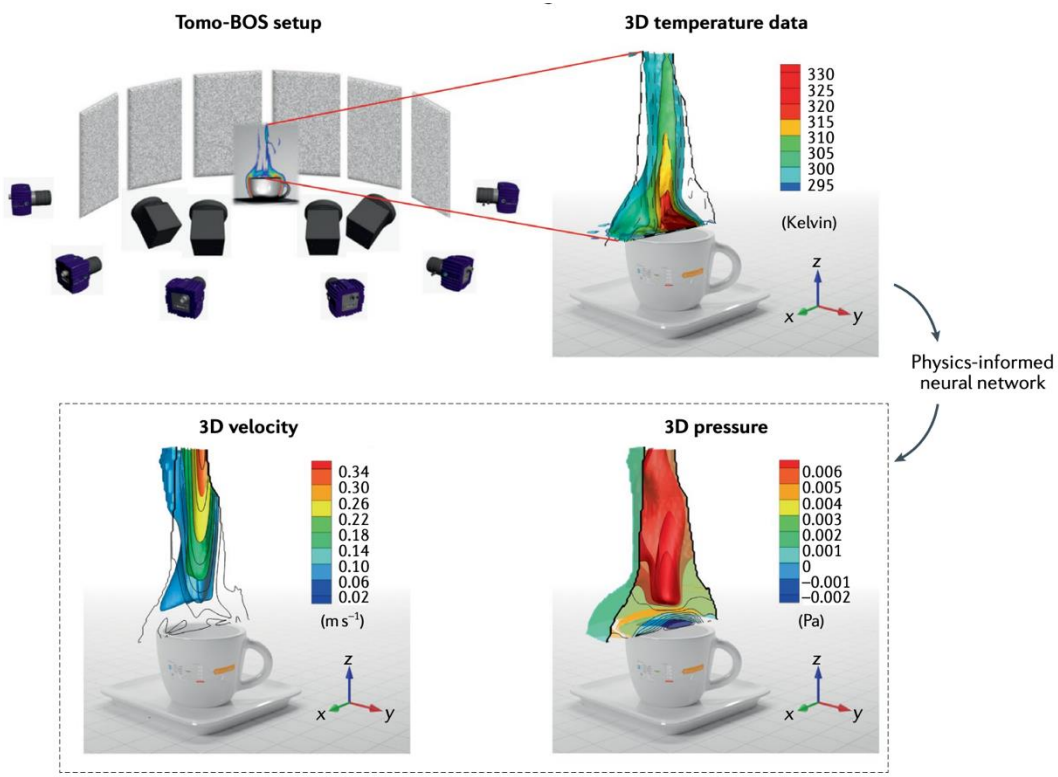
Exercise 0: PINN for mass-spring system

- Optional exercise, for the break or later
- Train a standard NN model and a PINN model for a (damped) mass-spring problem
- Learn both a damping coefficient and the spring coefficient

```python
def train_pinn(model, t_train, x_train, t_phys, nepochs=10000, learning_rate=5e-3):
    lambda_ = 1e-1
    mse = nn.MSELoss()
    t_phys = torch.tensor(t_phys, requires_grad=True, dtype=torch.float32).reshape(-1, 1)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    losses = []

    with trange(nepochs, desc="Training PINN with parameters") as pbar:
        for i in range(nepochs):
            optimizer.zero_grad()

            # Compute data loss
            x_pred = model(t_train)
            loss = mse(x_pred, x_train)

            # Compute physics loss (parameters mu and k in the ODE)
            mu = model.mu
            k = model.k
            x_phys = model(t_phys)
            xdot = time_derivative(x_phys, t_phys)
            xddot = time_derivative(xdot, t_phys)
            ode_residual = xddot + mu * xdot + k * x_phys
            loss += lambda_ * torch.mean(ode_residual**2)

            # Backward pass and parameter update
            loss.backward()
            optimizer.step()
```

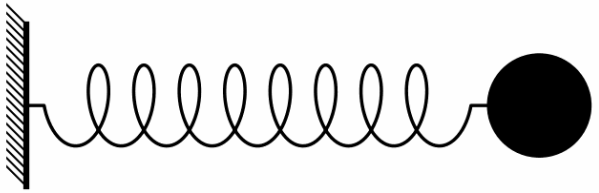
Why use PINN?

- General and easy to implement
  - Automatic differentiation to find derivatives
  - Almost as simple for PDEs as for ODEs
- Incomplete models and imperfect data
  - Can concentrate the data-driven learning on the areas where it is necessary
- Interpretable models
  - The physics term in the loss function has meaning
- Tackling high dimensionality
  - Breaks the curse of dimensionality
- Difficult geometries where meshing is not feasible
  - PINNs are mesh-free, as opposed to classical numerical methods



Cai et al. "Flow over an espresso cup: inferring 3-D velocity and pressure fields from tomographic background oriented Schlieren via physics-informed neural networks." *Journal of Fluid Mechanics* 915 (2021): A102.

Mass-spring system



Consider a simple mass-spring system, *without* damping or external forces. The force from the spring is $F = -k(x - x_0)$. Together with Newton's law $F = m\ddot{x}$, you get ($x_0 = 0$)

$$m\ddot{x} + kx = 0.$$

With $q = x$ and $p = m\dot{x}$, the total energy is

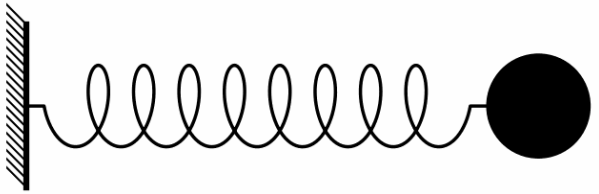
$$\mathcal{H}(q, p) = T(p) + V(q) = \frac{1}{2m}p^2 + \frac{k}{2}q^2,$$

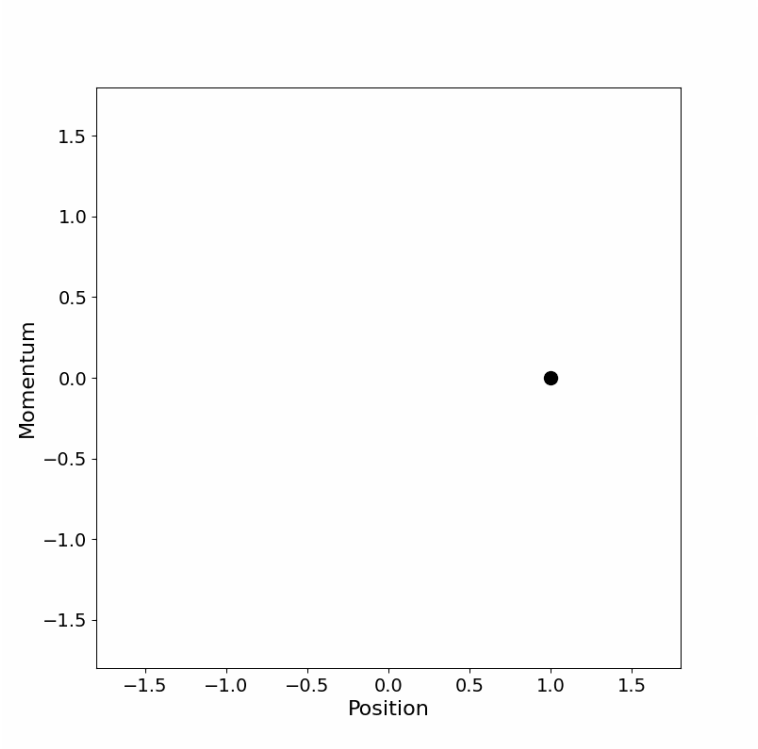
and Hamilton's equations

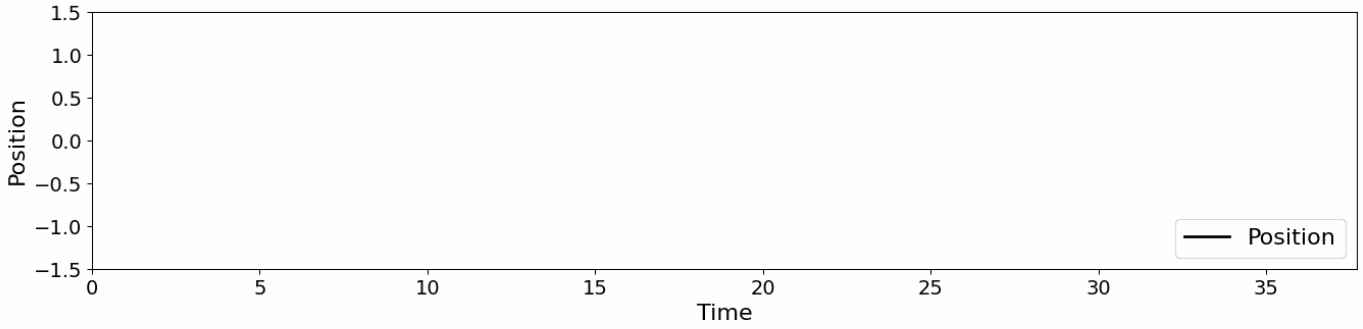
$$\dot{q} = \frac{\partial \mathcal{H}}{\partial p} = \frac{1}{m}p, \qquad \dot{p} = -\frac{\partial \mathcal{H}}{\partial q} = -kq$$

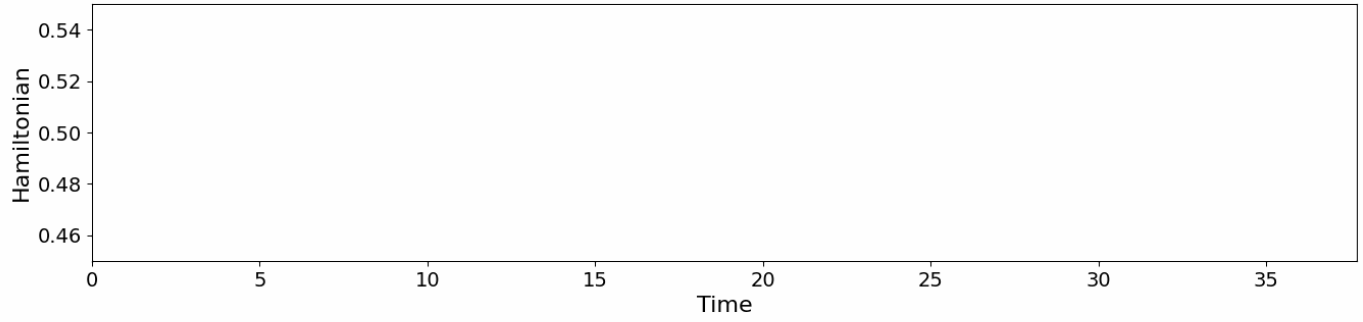
are equivalent to $m\ddot{q} + kq = 0$.

Mass-spring system

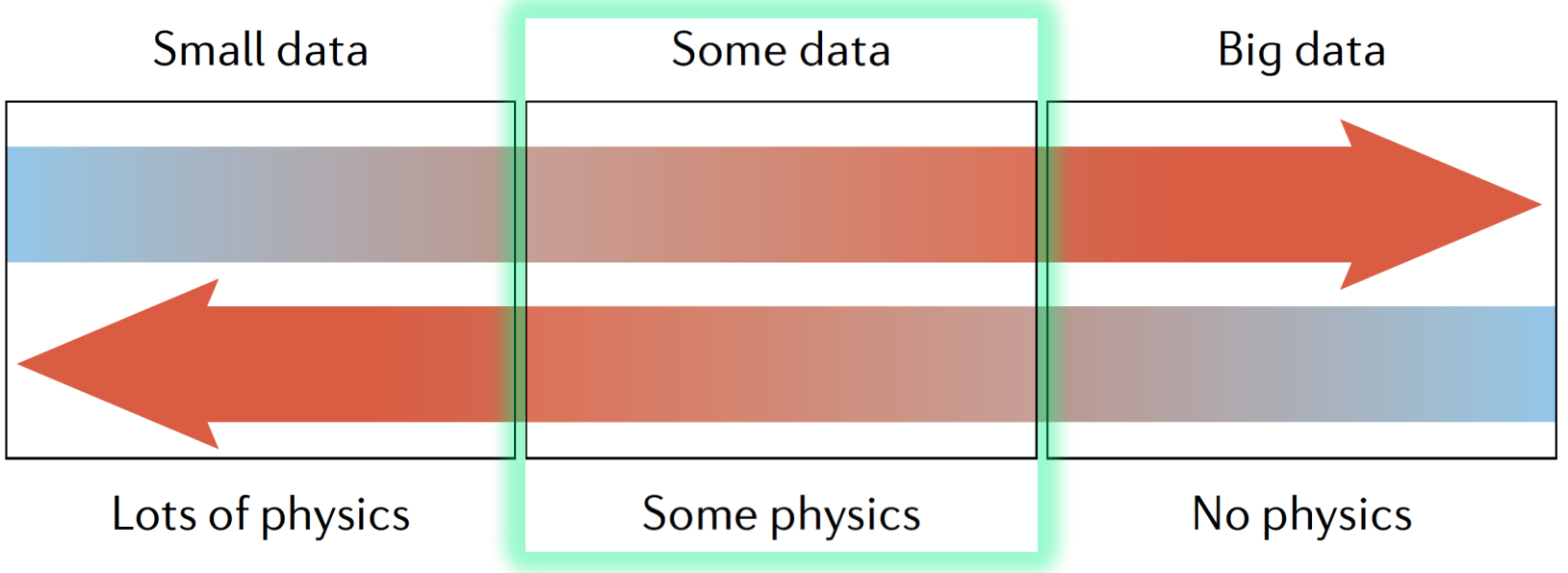








Mass-spring system



Know $m\ddot{x} + kx = 0$, but not m and/or k,
or know

$$\begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{H}}{\partial q} \\ \frac{\partial \mathcal{H}}{\partial p} \end{bmatrix}$$

but not $\mathcal{H}$
→ Use physics-informed ML

Part 3:

Hamiltonian neural networks

Hamiltonian neural networks (HNN)

Assuming a canonical Hamiltonian system, full knowledge of the Hamiltonian $\mathcal{H}$ gives full knowledge of the system.

We can use a neural network to model the Hamiltonian:

$$\hat{\mathcal{H}}(q, p) = \mathcal{NN}_\theta(q, p)$$

But

- We do not know the Hamiltonian since it is not observed
- If we would observe the energy, it would not be unique ($\mathcal{H}(q, p) = \mathcal{H}(q', p'), x \neq x'$)

Hamiltonian neural networks

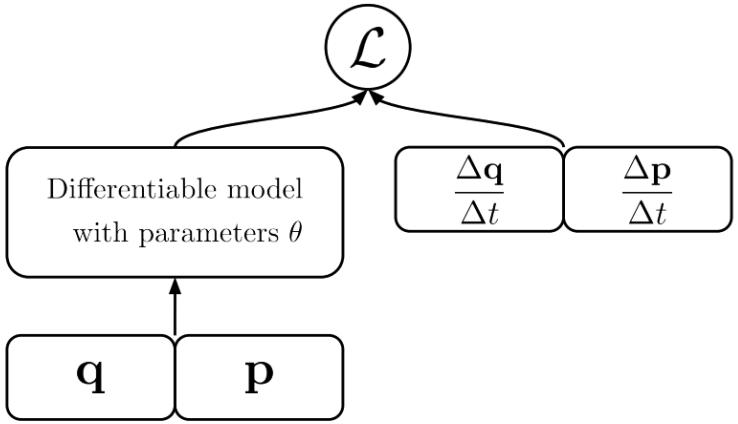
We can compute gradients w.r.t. input of a neural net. Thus

$$\hat{\dot{q}} = \frac{\partial \hat{\mathcal{H}}(q,p)}{\partial p}, \qquad \hat{\dot{p}} = -\frac{\partial \hat{\mathcal{H}}(q,p)}{\partial q}$$

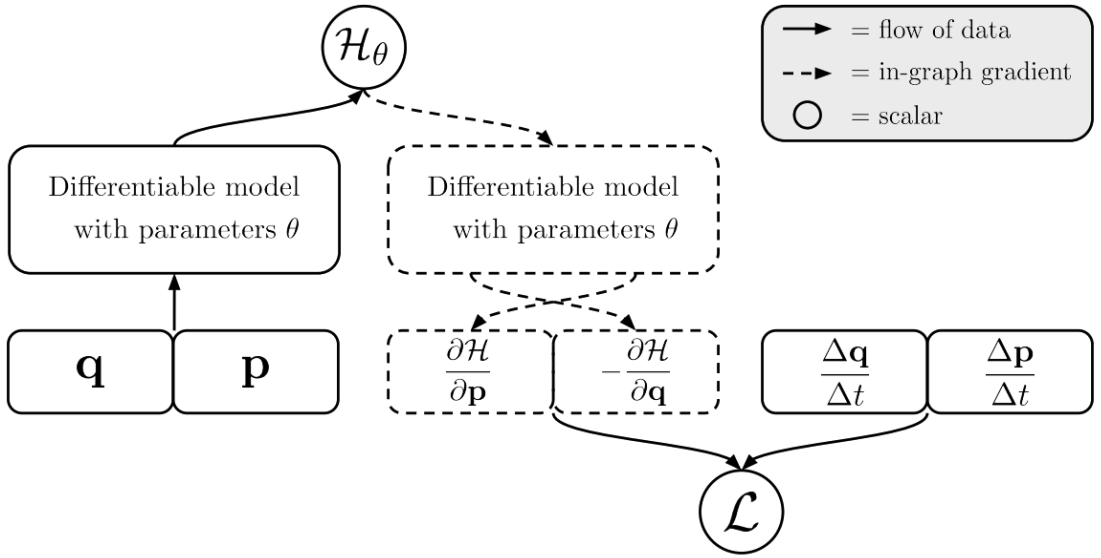
and the loss function

$$\mathcal{L} = \left|\dot{q} - \hat{\dot{q}}\right|_2 + \left|\dot{p} - \hat{\dot{p}}\right|_2$$

Hamiltonian neural networks



(a) Baseline NN

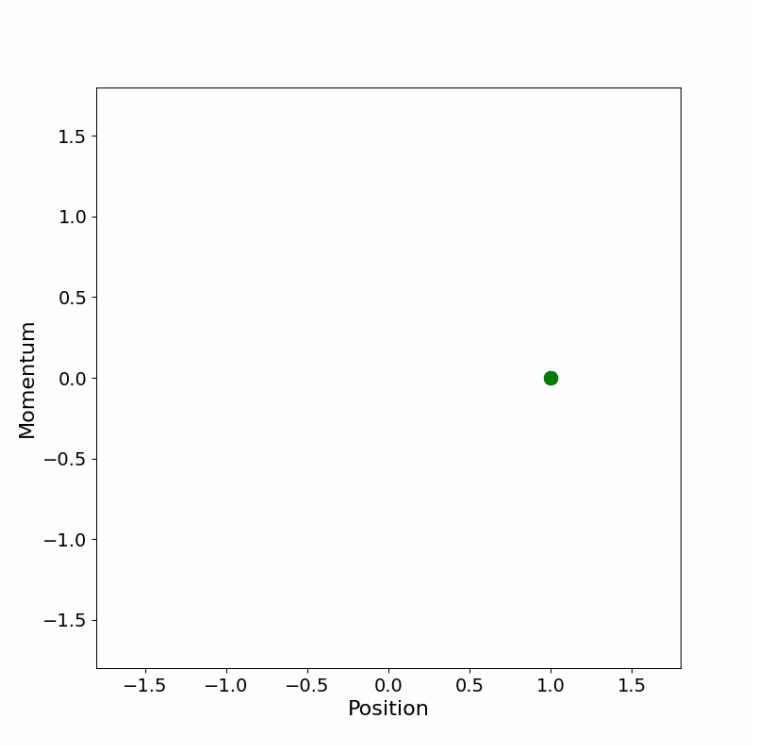


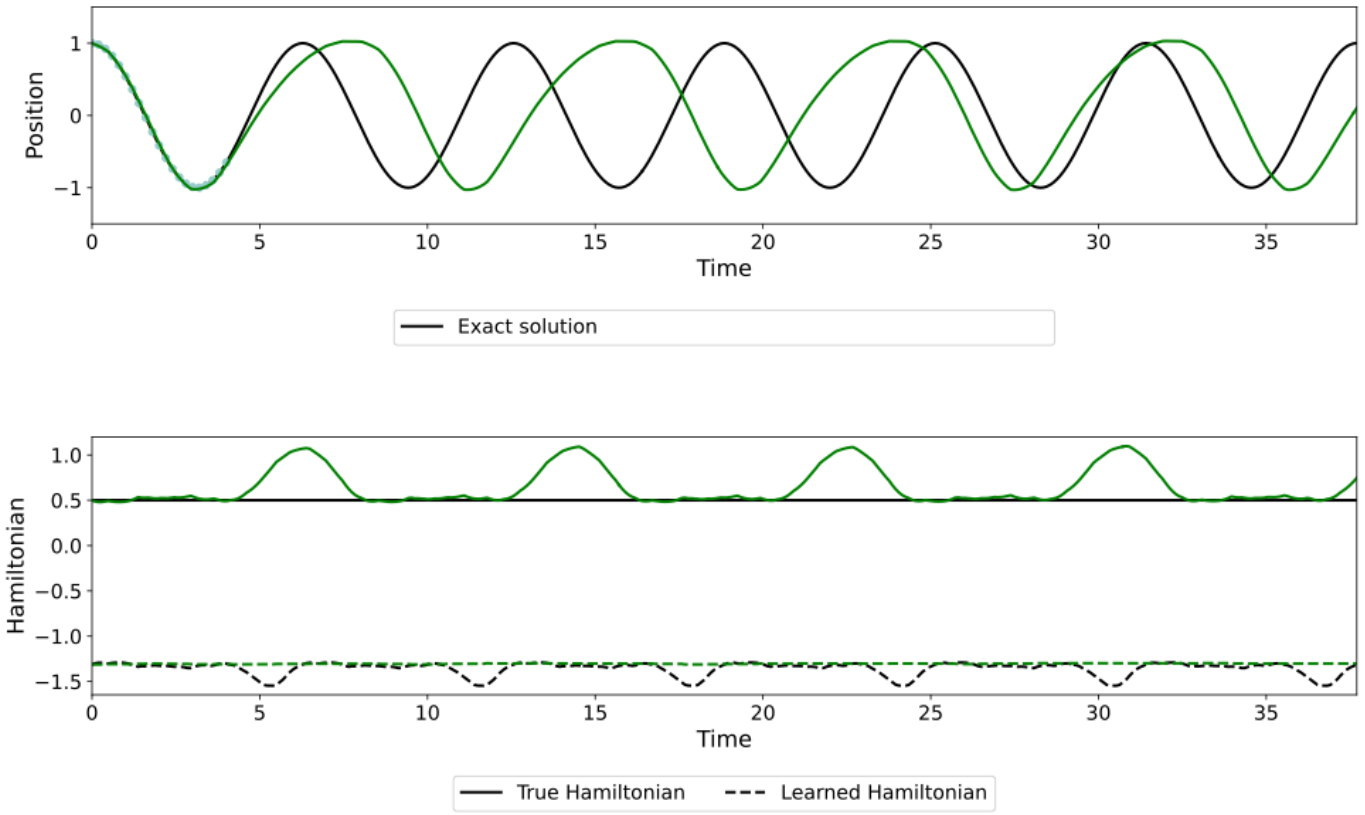
(b) Hamiltonian NN

Greydanus, Dzamba, and Yosinski. "Hamiltonian neural networks." *Advances in neural information processing systems* 32 (2019).

HNN on mass-spring system

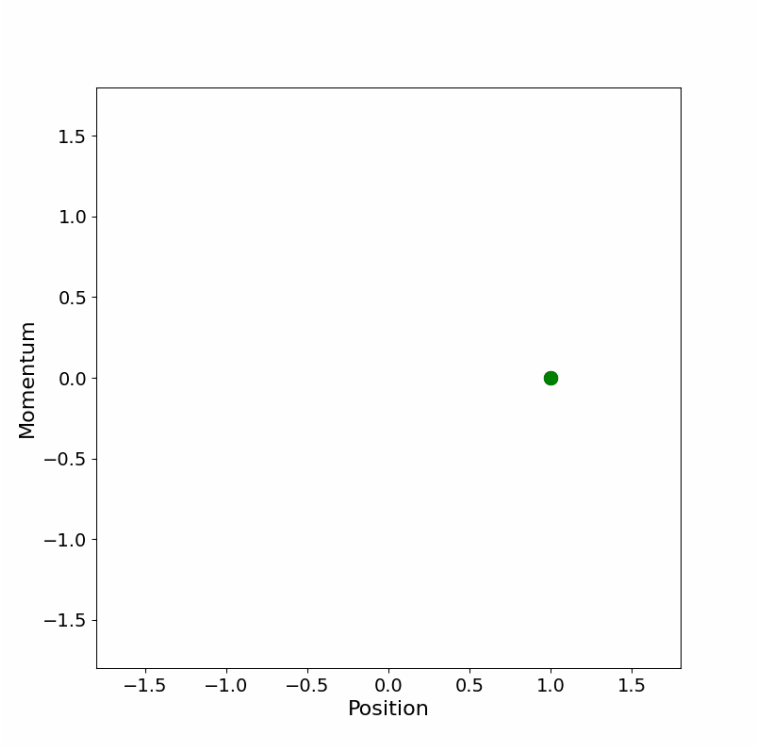
After 200 epochs:

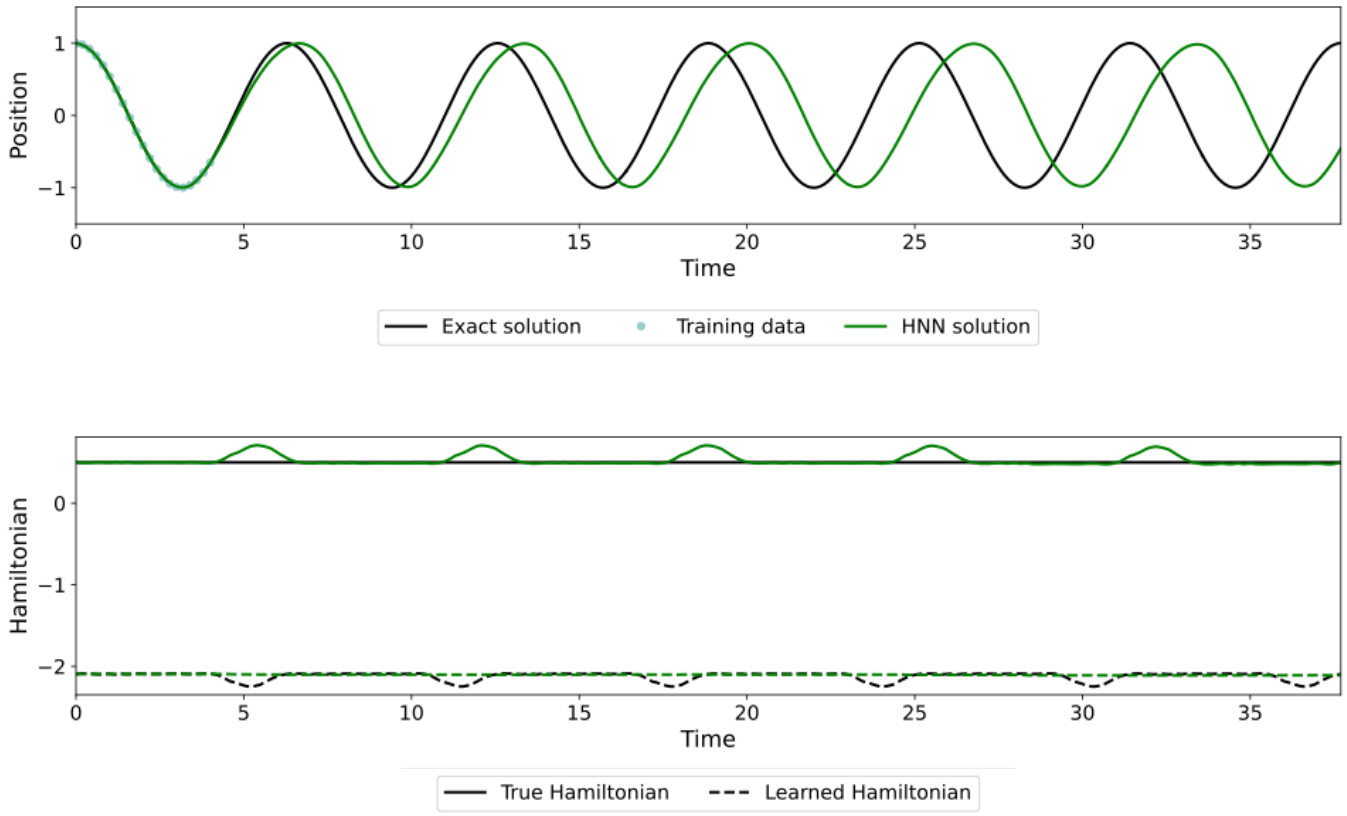




HNN on mass-spring system

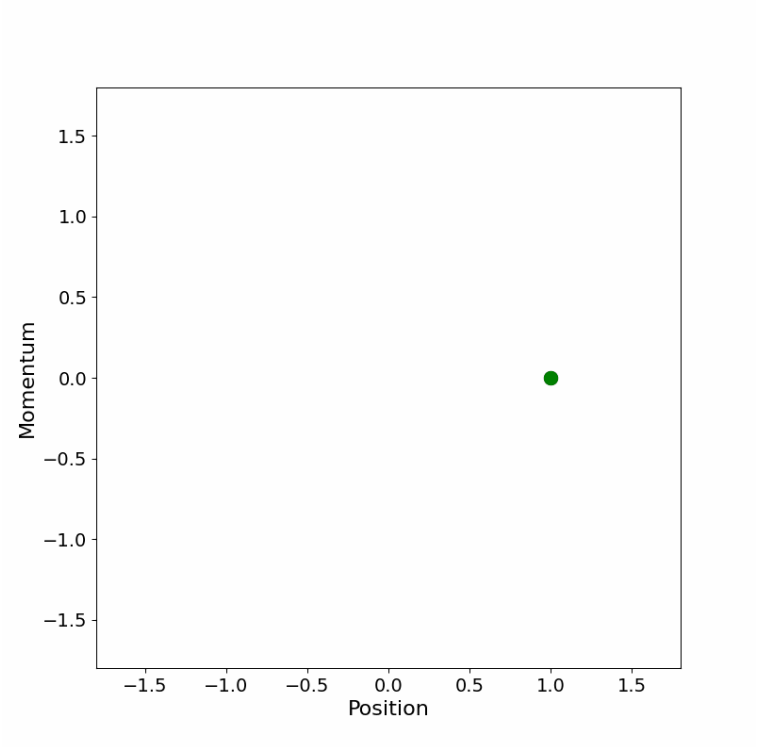
After 1000 epochs:

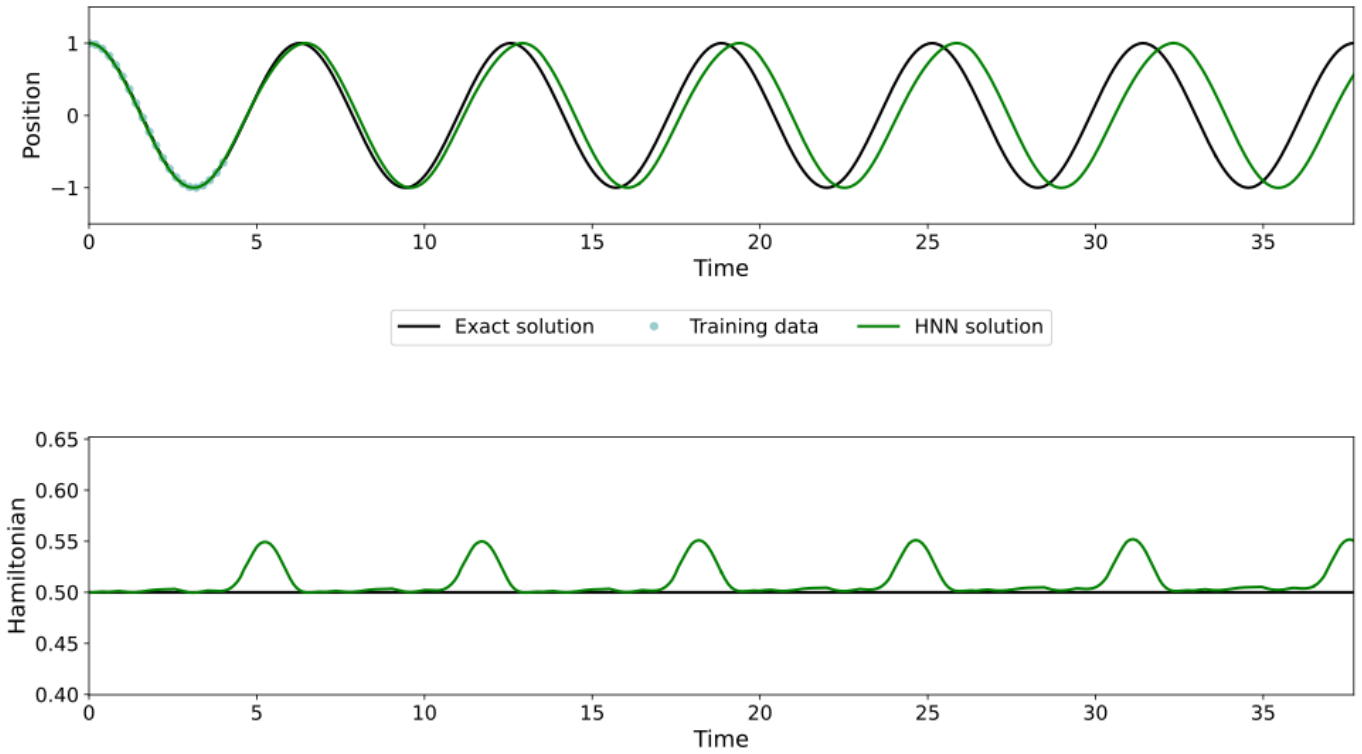




HNN on mass-spring system

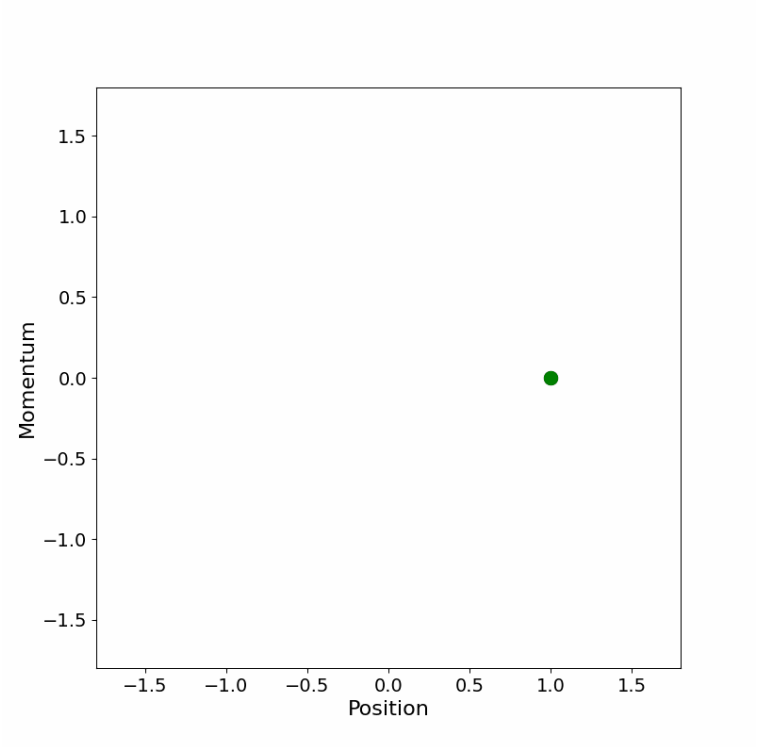
After 3000 epochs:

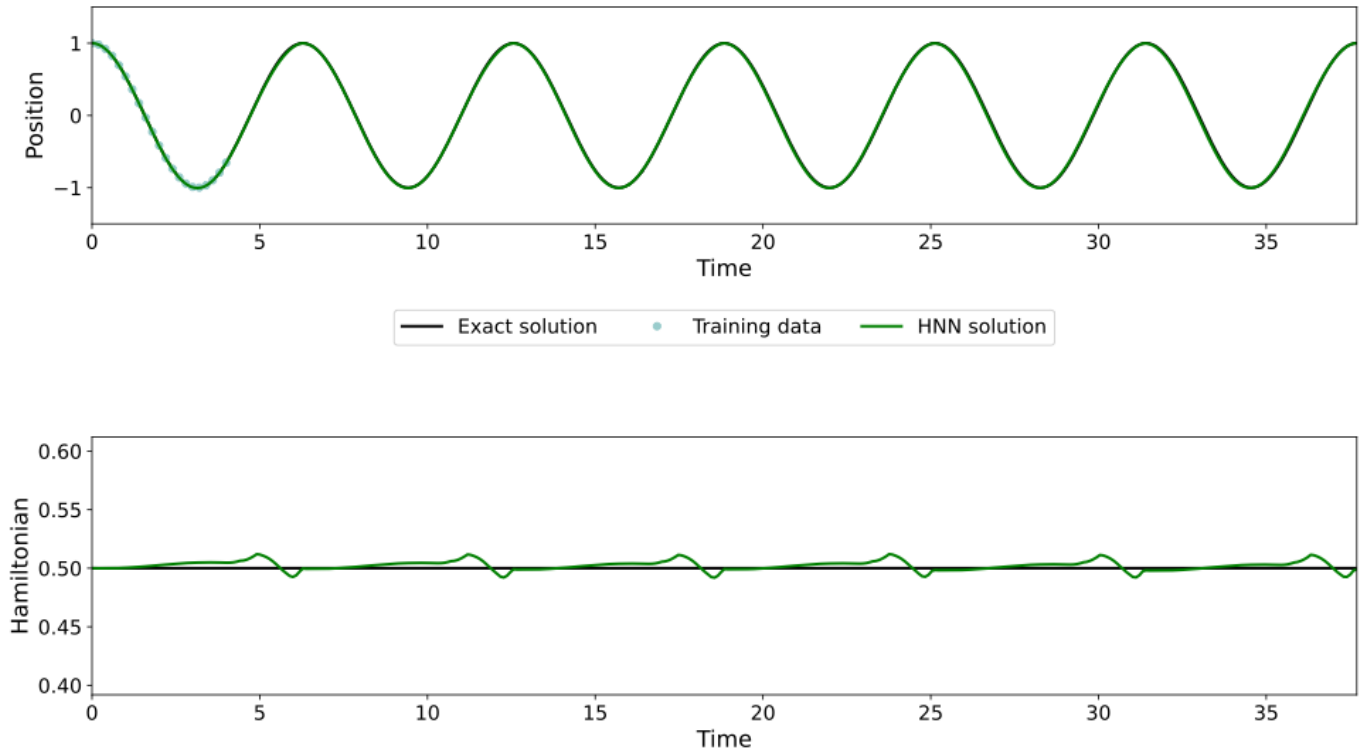




HNN on mass-spring system

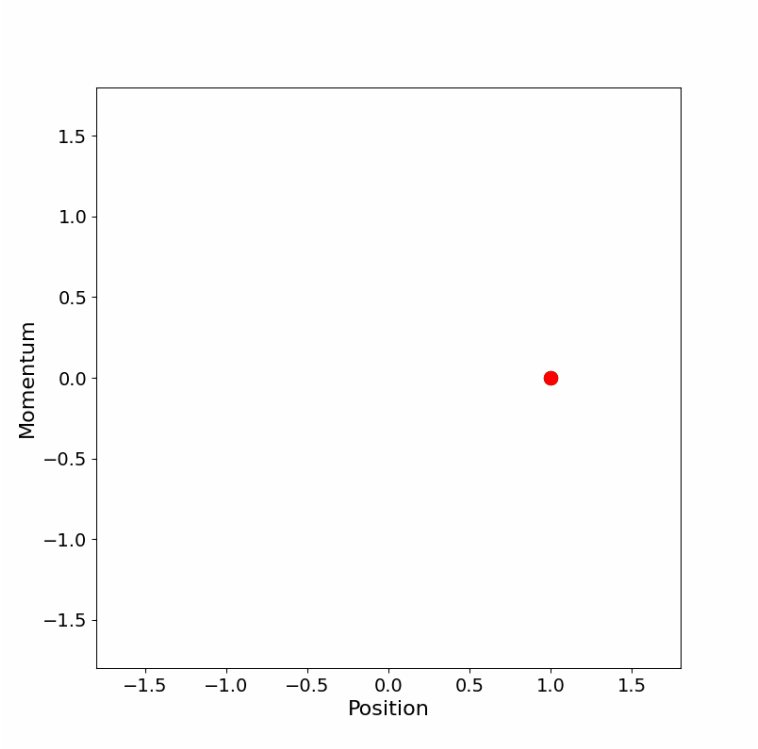
After 20 000 epochs:

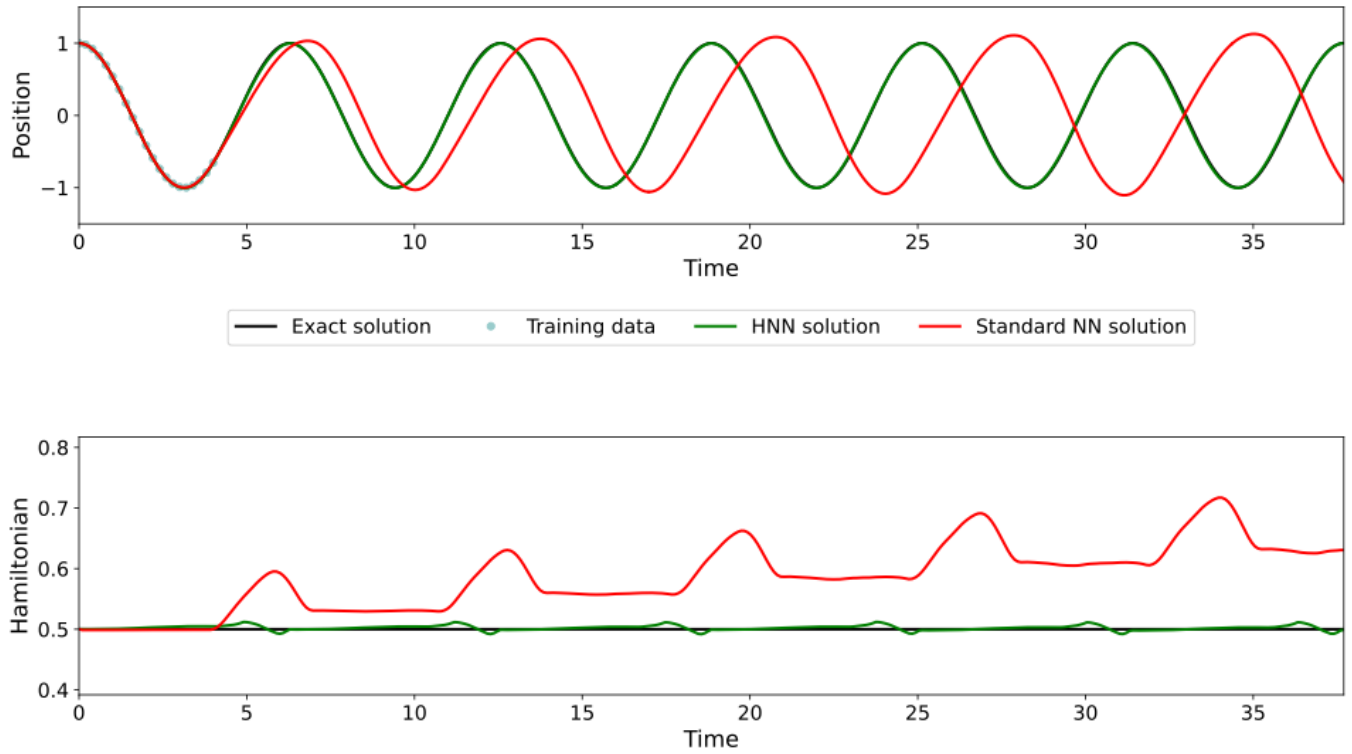




HNN on mass-spring system

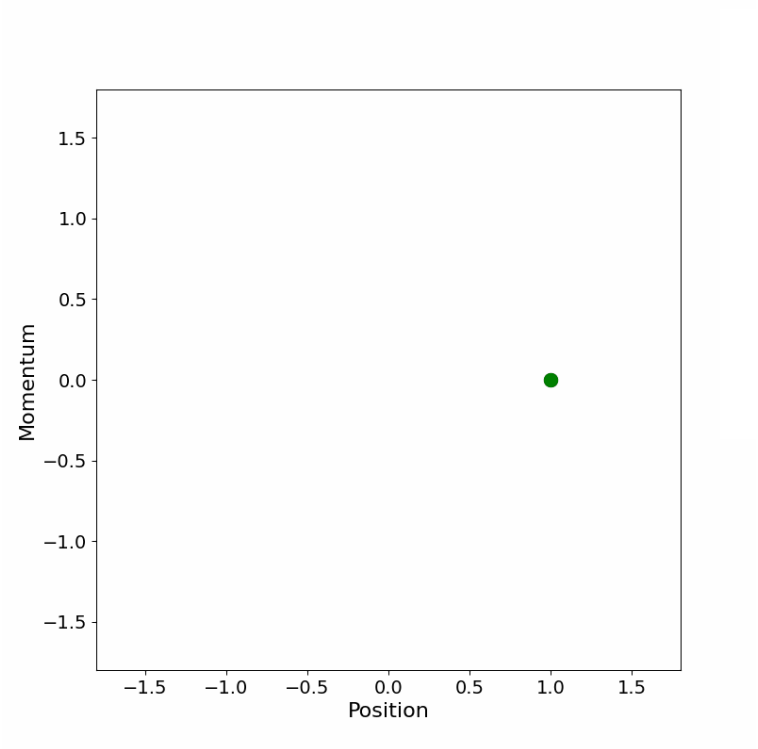
After 20 000 epochs:

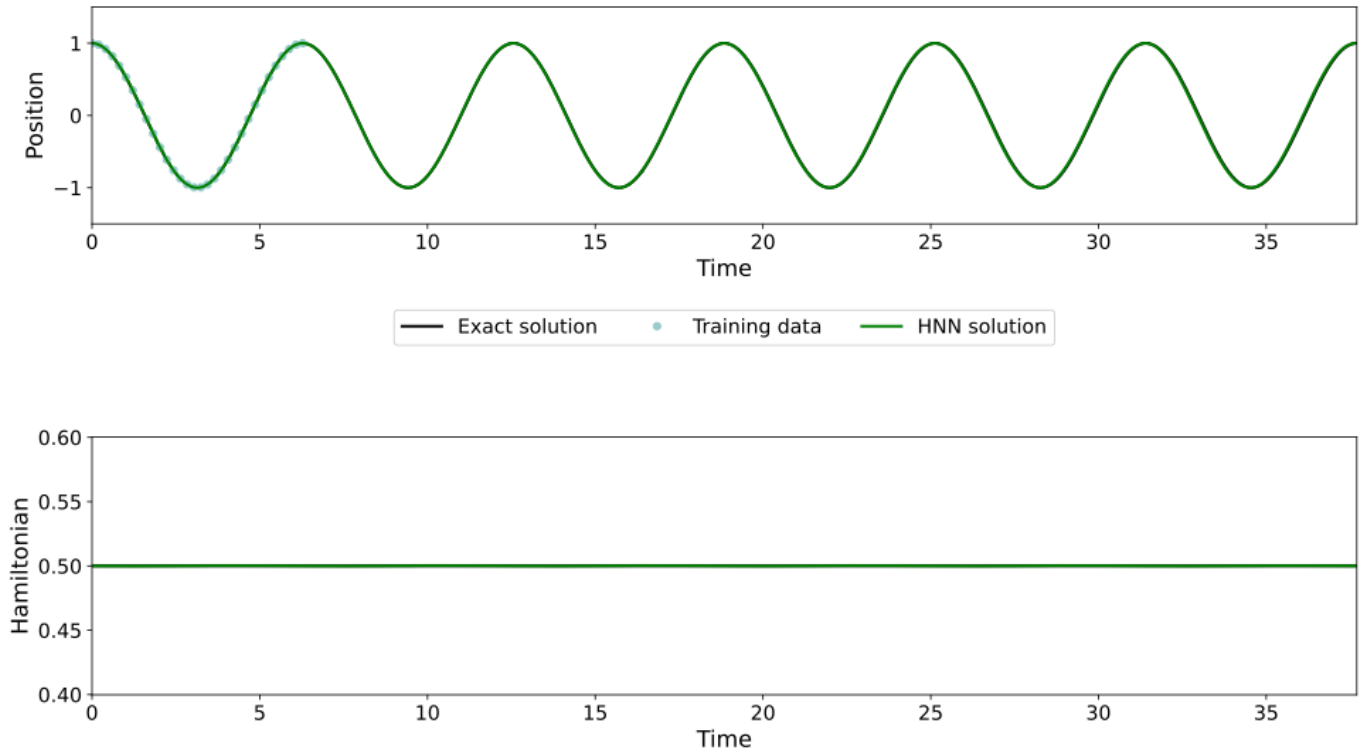




HNN on mass-spring system

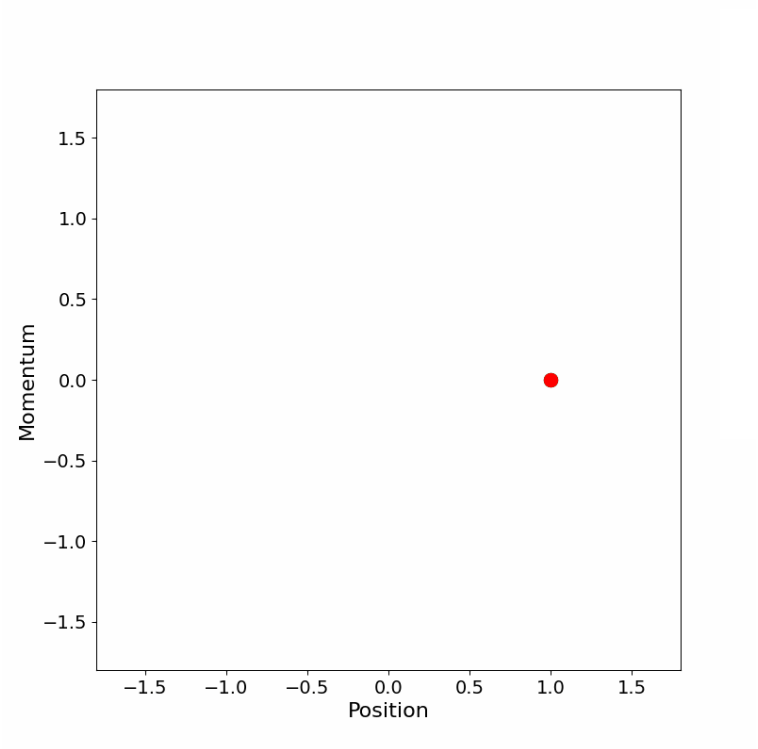
After 20 000 epochs:

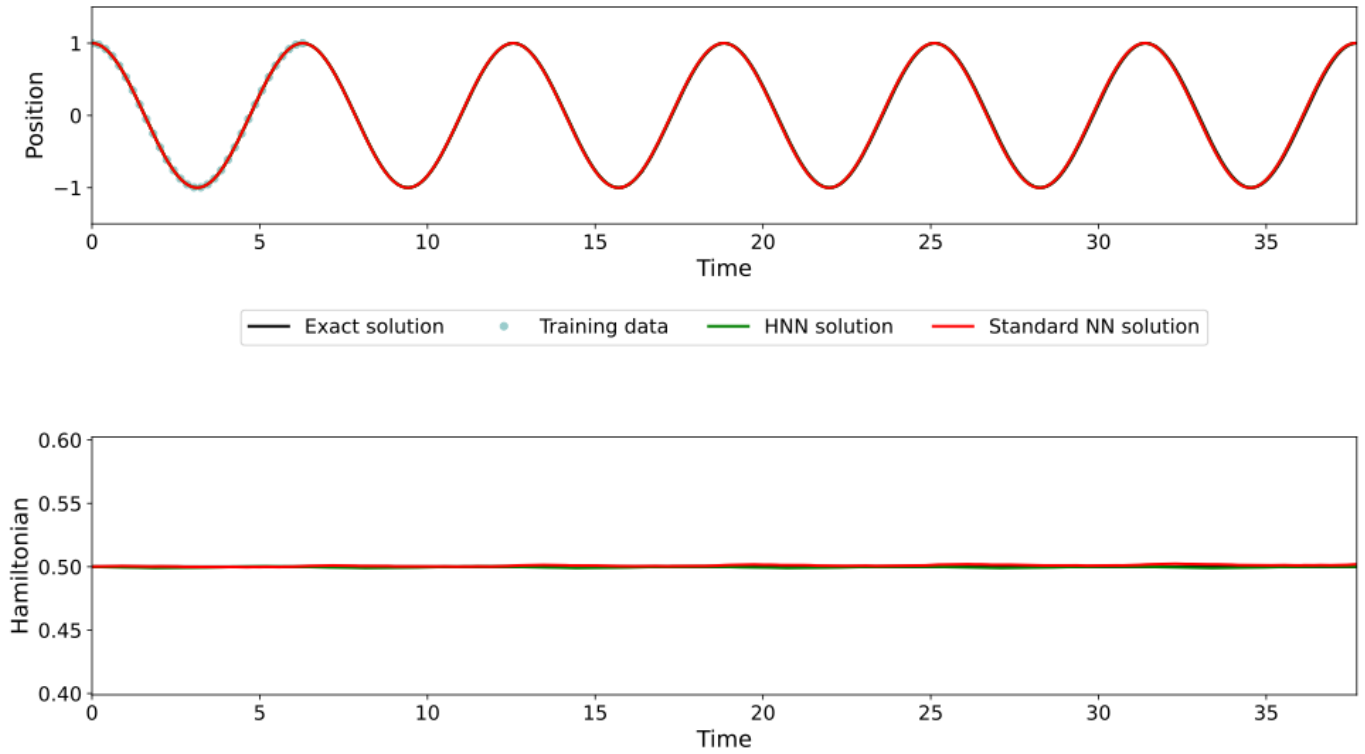




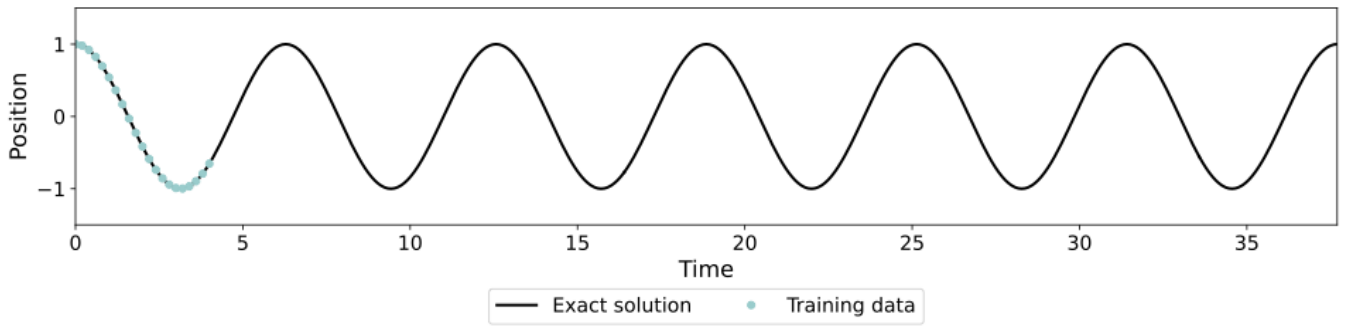
HNN on mass-spring system

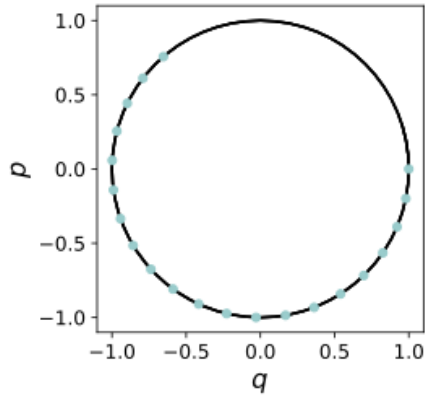
After 20 000 epochs:

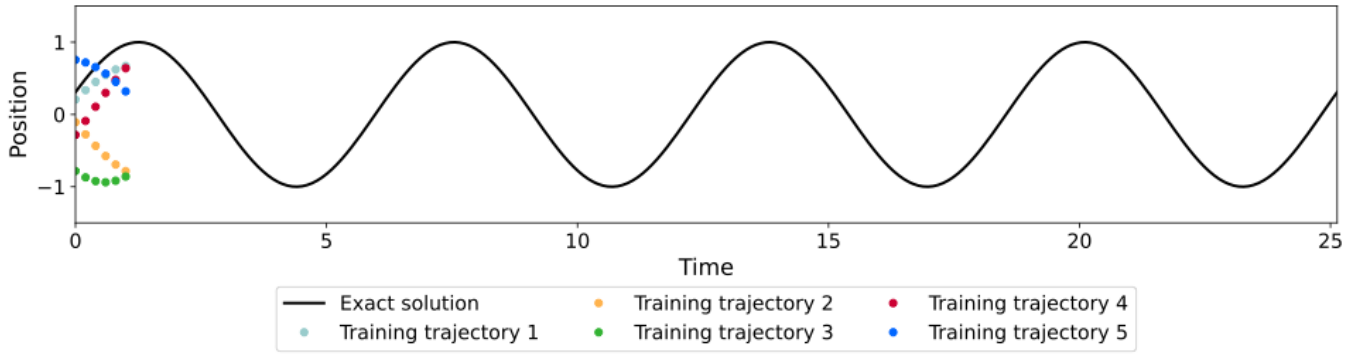


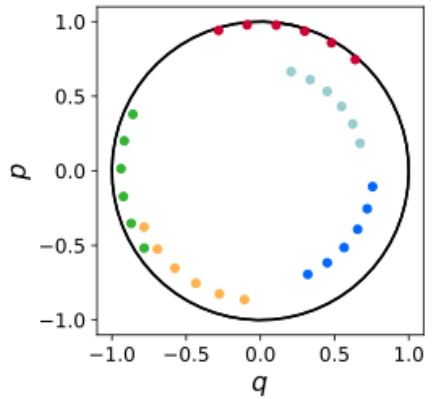


HNN on mass-spring system









Why use structure-preserving models?

Similar as for structure-preserving integrators:

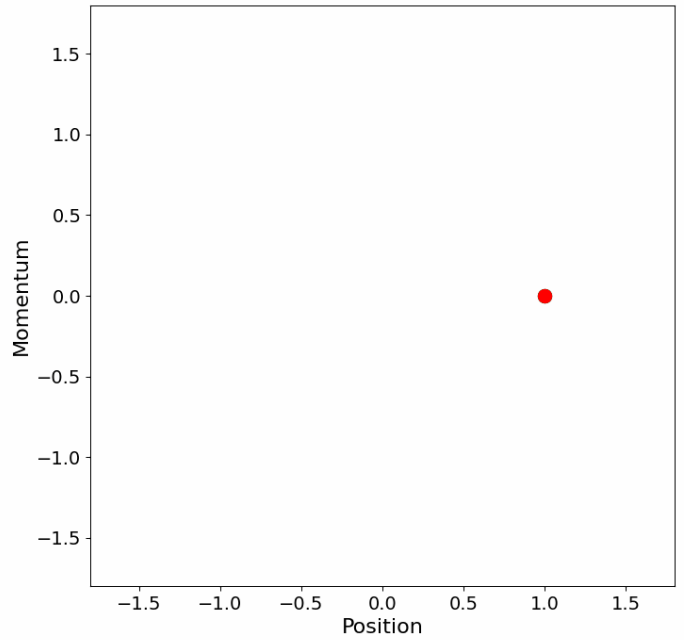
- Good long-term behaviour
- Stability guarantees
- The structure-preservation itself

In addition:

- A little less black box

Compared to PINN:

- Hard constraints
- Does not require user-defined collocation points
- Simpler loss function – more efficient backpropagation



Part 4:

Pseudo-Hamiltonian neural networks for ODEs

Pseudo-Hamiltonian neural networks (PHNN)

Consider a mass-spring system with damping and external forces:

$$m\ddot{x} + c\dot{x} + kx = f(x,t),$$

where

- m is the mass,
- c is the damping coefficient,
- k is the stiffness coefficient,
- f is the force term.

With $q = x$, $p = m\dot{x}$ and $\mathcal{H}(q,p) = \frac{1}{2m}p^2 + \frac{k}{2}q^2$, this is equivalent to

$$\begin{bmatrix}\dot{q}\\ \dot{p}\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & -c\end{bmatrix}\right)\begin{bmatrix}\frac{\partial \mathcal{H}}{\partial q}(q,p)\\ \frac{\partial \mathcal{H}}{\partial p}(q,p)\end{bmatrix} + \begin{bmatrix}0\\ f(q,p,t)\end{bmatrix}.$$

Pseudo-Hamiltonian neural networks (PHNN)

Consider a mass-spring system with damping and external forces:

$$m\ddot{x} + c\dot{x} + kx = f(x,t),$$

where

- m is the mass,
- c is the damping coefficient,
- k is the stiffness coefficient,
- f is the force term.

With $q = x$, $p = m\dot{x}$ and $\mathcal{H}(q,p) = \frac{1}{2m}p^2 + \frac{k}{2}q^2$, this is equivalent to

$$\begin{bmatrix}\dot{q}\\ \dot{p}\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & -\hat{c}\end{bmatrix}\right)\begin{bmatrix}\frac{\partial\hat{\mathcal{H}}_\theta}{\partial q}(q,p)\\ \frac{\partial\hat{\mathcal{H}}_\theta}{\partial p}(q,p)\end{bmatrix} + \begin{bmatrix}0\\ \hat{f}_\theta(q,p,t)\end{bmatrix}.$$

Pseudo-Hamiltonian neural networks (PHNN)

Consider a mass-spring system with damping and external forces:

$$m\ddot{x} + c\dot{x} + kx = f(x,t),$$

where

- m is the mass,
- c is the damping coefficient,
- k is the stiffness coefficient,
- f is the force term.

With $q = x$, $p = m\dot{x}$ and $\mathcal{H}(q,p) = \frac{1}{2m}p^2 + \frac{k}{2}q^2$, this is equivalent to

$$L = \left\| \begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} - \left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -\hat{c} \end{bmatrix} \right) \begin{bmatrix} \frac{\partial \hat{\mathcal{H}}_\theta}{\partial q}(q,p) \\ \frac{\partial \hat{\mathcal{H}}_\theta}{\partial p}(q,p) \end{bmatrix} + \begin{bmatrix} 0 \\ \hat{f}_\theta(q,p,t) \end{bmatrix} \right\|_2^2$$

Pseudo-Hamiltonian neural networks

$$L = \left\| \begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} - \left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -\hat{c} \end{bmatrix} \right) \begin{bmatrix} \frac{\partial \hat{\mathcal{H}}_\theta}{\partial q}(q,p) \\ \frac{\partial \hat{\mathcal{H}}_\theta}{\partial p}(q,p) \end{bmatrix} + \begin{bmatrix} 0 \\ \hat{f}_\theta(q,p,t) \end{bmatrix} \right\|_2^2$$

In general, we assume

$$\dot{x} = g(x,t)$$

and train a model

$$\hat{g}_\theta(x,t) := (S - \hat{R}_\theta)\nabla \hat{\mathcal{H}}_\theta(x) + \hat{f}_\theta(x,t)$$

for $g(x,t)$, where S is skew-symmetric and $\hat{R}_\theta$ is positive semi-definite.

Exercise 1: PHNN for the mass-spring system

- Set up and train a standard NN model to learn the dynamics of the system by learning the right-hand side of the ODE
  - What should the input and output dimensions of the NN be?
- Set up and train a HNN model to learn a mass-spring system
  - What should the input and output dimension of the NN be?
- Set up and train a PHNN model to learn a damped mass-spring system
  - Make the spring coefficient trainable

```
model_snn = FNN(SET_INPUT_DIM, 32, SET_OUTPUT_DIM)
```

```
class PHNN(nn.Module):
    def __init__(self, input_dim=SET_INPUT_DIM, hidden_dim=32):
        super().__init__()
        self.hamiltonian_net = FNN(input_size=input_dim, hidden_size=hidden_dim, output_size=SET_OUTPUT_DIM)
        self.mu = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.S = torch.tensor([[0.0, 1.0], [-1.0, 0.0]])

    def hamiltonian(self, x):
        return self.hamiltonian_net(x)

    def grad(self, x):
        return torch.autograd.grad(self.hamiltonian(x).sum(), x, create_graph=True)[0]

    def forward(self, x):
        S = self.S.clone().detach()
        S[1, 1] = -self.mu
        dH = self.grad(x)
        return (S @ dH.T).T
```

Part 5:

Pseudo-Hamiltonian neural networks for PDEs

PHNN for learning PDEs

We consider the class of pseudo-Hamiltonian PDEs

$$u_t = S(u^J, x)\frac{\delta \mathcal{H}}{\delta u}[u] - R(u^J, x)\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t),$$

where

- $S(u^J, x)$ is a skew-symmetric operator (w.r.t. the L_2 inner product)
- $R(u^J, x)$ is a positive semidefinite operator (w.r.t. the L_2 inner product)
- $\mathcal{H}$ and $\mathcal{V}$ are integrals over the spatial domain
- $\frac{\delta \mathcal{H}}{\delta u}$ is the variational derivative of $\mathcal{H}$

PHNN for learning PDEs

We consider the class of pseudo-Hamiltonian PDEs

$$u_t = S(u^J, x)\frac{\delta \mathcal{H}}{\delta u}[u] - R(u^J, x)\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t),$$

where

- $S(u^J, x)$ is a skew-symmetric operator (w.r.t. the L_2 inner product)
- $R(u^J, x)$ is a positive semidefinite operator (w.r.t. the L_2 inner product)
- $\mathcal{H}$ and $\mathcal{V}$ are integrals over the spatial domain
- $\frac{\delta \mathcal{H}}{\delta u}$ is the variational derivative of $\mathcal{H}$

PHNN for learning PDEs

We consider the class of pseudo-Hamiltonian PDEs

$$u_t = \hat{S}_\theta(u^J, x)\frac{\delta \hat{\mathcal{H}}_\theta}{\delta u}[u] - \hat{R}_\theta(u^J, x)\frac{\delta \hat{\mathcal{V}}_\theta}{\delta u}[u] + \hat{f}_\theta(u^J, x, t),$$

where

- $S(u^J, x)$ is a skew-symmetric operator (w.r.t. the L_2 inner product)
- $R(u^J, x)$ is a positive semidefinite operator (w.r.t. the L_2 inner product)
- $\mathcal{H}$ and $\mathcal{V}$ are integrals over the spatial domain
- $\frac{\delta \mathcal{H}}{\delta u}$ is the variational derivative of $\mathcal{H}$

Pseudo-Hamiltonian PDEs

$$u_t = S(u^J, x)\frac{\delta \mathcal{H}}{\delta u}[u] - R(u^J, x)\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t)$$

Example: The forced KdV–Burgers (or viscous KdV) equation

$$u_t + \eta u u_x + \gamma^2 u_{xxx} - \nu u_{xx} = f(u^J, x, t),$$

where

$$S(u^J, x) = \frac{\partial}{\partial x}, \quad \mathcal{H} = \int_{\mathbb{R}} \left(-\frac{\eta}{6}u^3 + \frac{\gamma}{2}u_x^2\right) dx$$

and

$$R(u^J, x) = I, \quad \mathcal{V} = \frac{\nu}{2}\int_{\mathbb{R}} u_x^2 \, dx.$$

Pseudo-Hamiltonian PDEs

We assume to have the discretized first integral $\mathcal{H}_{\mathbf{p}}(\mathbf{u})$ and approximate $S(\mathbf{x}, u^J)$ by $S_d(\mathbf{u})$, skew-symmetric w.r.t. the discrete inner product $\langle \mathbf{u}, \mathbf{v} \rangle_\kappa := \sum_{i=0}^{M} \kappa_i u_i v_i$. Then we define the discrete analogue to the variational derivative by

$$\left\langle \frac{\delta \mathcal{H}_{\mathbf{p}}}{\delta \mathbf{u}}(\mathbf{u}), \mathbf{v} \right\rangle_\kappa = \frac{\mathrm{d}}{\mathrm{d}\epsilon}\bigg|_{\epsilon=0} \mathcal{H}_{\mathbf{p}}(\mathbf{u} + \epsilon \mathbf{v}) \quad \forall \mathbf{v} \in \mathbb{R}^{M+1}$$

and have that

$$\frac{\delta \mathcal{H}_{\mathbf{p}}}{\delta \mathbf{u}}(\mathbf{u}) = \mathrm{diag}(\kappa)^{-1} \nabla_{\mathbf{u}} \mathcal{H}_{\mathbf{p}}(\mathbf{u}).$$

Also, we let $S_{\mathbf{p}}(\mathbf{u}) := S_d(\mathbf{u})\, \mathrm{diag}(\kappa)^{-1}$.

Pseudo-Hamiltonian PDEs

Thus a spatial discretization of

$$u_t = S(u^J, x)\frac{\delta \mathcal{H}}{\delta u}[u] - R(u^J, x)\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t)$$

is given by

$$\mathbf{u}_t = S_d(\mathbf{u})\frac{\delta \mathcal{H}_\mathbf{p}}{\delta \mathbf{u}}(\mathbf{u}) - R_d(\mathbf{u})\frac{\delta \mathcal{V}_\mathbf{p}}{\delta \mathbf{u}}(\mathbf{u}) + \mathbf{f}(\mathbf{u}, \mathbf{x}, t),$$

which may equivalently be written as

$$\mathbf{u}_t = S_\mathbf{p}(\mathbf{u})\nabla_\mathbf{u}\mathcal{H}_\mathbf{p}(\mathbf{u}) - R_\mathbf{p}(\mathbf{u})\nabla_\mathbf{u}\mathcal{V}_\mathbf{p}(\mathbf{u}) + \mathbf{f}(\mathbf{u}, \mathbf{x}, t).$$

If $\mathcal{V}_\mathbf{p} = \mathcal{H}_\mathbf{p}$, this is a pseudo-Hamiltonian ODE system

$$\dot{\mathbf{u}} = (S(\mathbf{u}) - R(\mathbf{u}))\nabla H(\mathbf{u}) + f(\mathbf{u}, t).$$

Pseudo-Hamiltonian PDEs

Recall: The forced KdV-Burgers equation

$$u_t + \eta u u_x + \gamma^2 u_{xxx} - \nu u_{xx} = f(u^J, x, t)$$

is of the form

$$Au_t = S\frac{\delta \mathcal{H}}{\delta u}[u] - R\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t)$$

with $A = I$, $S = \frac{\partial}{\partial x}$ and $R = I$, and

$$\mathcal{H} = \int_{\mathbb{R}} \left(-\frac{\eta}{6}u^3 + \frac{\gamma}{2}u_x^2\right) \mathrm{d}x, \quad \mathcal{V} = \int_{\mathbb{R}} \frac{\nu}{2}u_x^2 \, dx.$$

Pseudo-Hamiltonian PDEs

Recall: The forced KdV-Burgers equation

$$u_t + \eta u u_x + \gamma^2 u_{xxx} - \nu u_{xx} = f(u^J, x, t)$$

is of the form

$$Au_t = S\frac{\delta \mathcal{H}}{\delta u}[u] - R\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t)$$

with $A = I$, $S = \frac{\partial}{\partial x}$ and $R = I$, and

$$\mathcal{H} = \int_{\mathbb{R}} \left(-\frac{\eta}{6}u^3 + \frac{\gamma}{2}u_x^2\right) \mathrm{d}x, \quad \mathcal{V} = \int_{\mathbb{R}} \frac{\nu}{2}u_x^2 \, dx.$$

Pseudo-Hamiltonian PDEs

Approximate $\mathcal{H} = \int_{\mathbb{R}} \left(-\frac{\eta}{6}u^3 + \frac{\gamma}{2}u_x^2\right)\, dx$ and $\mathcal{V} = \frac{\nu}{2}\int_{\mathbb{R}} u_x^2\, dx$ by

$$\mathcal{H}_{\mathbf{p}} = -h\sum_{i=0}^{M-1}\left(\frac{\eta}{6}u_i^3 + \frac{\gamma}{2}\left(\delta_f u_i\right)^2\right), \quad \mathcal{V}_{\mathbf{p}} = \frac{\nu}{2}h\sum_{i=0}^{M-1}\kappa_i\left(\delta_f u_i\right)^2,$$

where $\delta_f u_i = \frac{u_{i+1}-u_i}{h}$. Then

$$\nabla_u \mathcal{H}_{\mathbf{p}} = -h\left(\frac{\eta}{2}\mathbf{u}^2 - \gamma^2\delta_c^2\mathbf{u}\right),$$
$$\nabla_u \mathcal{V}_{\mathbf{p}} = -h\,\nu\,\delta_c^2\mathbf{u},$$

where $\delta_c^2 u_i = \frac{1}{2h}(u_{i+1} - 2u_i + u_{i-1})$.

We have $\frac{\delta\mathcal{H}_{\mathbf{p}}}{\delta\mathbf{u}}(\mathbf{u}) = \frac{1}{h}\nabla_{\mathbf{u}}\mathcal{H}_{\mathbf{p}}(\mathbf{u})$ and $\frac{\delta\mathcal{V}_{\mathbf{p}}}{\delta\mathbf{u}}(\mathbf{u}) = \frac{1}{h}\nabla_{\mathbf{u}}\mathcal{V}_{\mathbf{p}}(\mathbf{u})$.

Pseudo-Hamiltonian PDEs

Discrete convolution of the function u and the kernel or filter w:

$$(u * w)(x_i) = \sum_{j=-m}^{m} w_j u(x_{i-j}).$$

A finite difference approximation of the n-th order derivative of u at a point x_i:

$$\frac{\mathrm{d}^n u(x_i)}{\mathrm{d}x^n} \approx \sum_{j=-m}^{m} a_j u(x_{i-j}),$$

E.g.,

$$\frac{\mathrm{d}u(x_i)}{\mathrm{d}x} = \frac{u(x_{i+1}) - u(x_{i-1})}{2h} + \mathcal{O}(h^2)$$

$$\frac{\mathrm{d}^2 u(x_i)}{\mathrm{d}x^2} = \frac{u(x_{i+1}) - u(x_i) + u(x_{i-1})}{h^2} + \mathcal{O}(h^2)$$

PHNN for PDEs

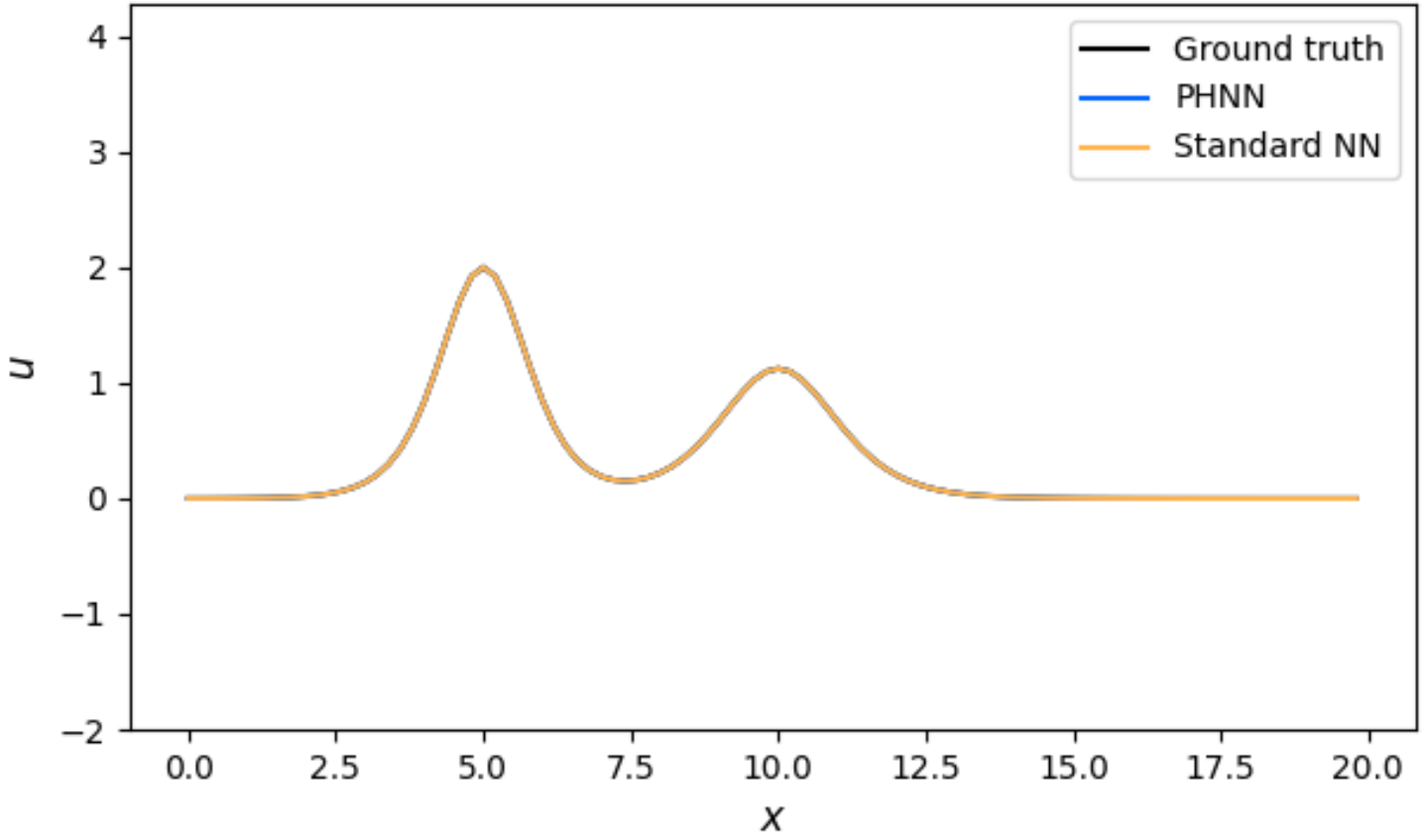
The full PHNN model for PDEs is given by

$$\hat{g}_\theta(u, x, t) = (\hat{A}_\theta^{[k_1]})^{-1}\left(\hat{S}_\theta^{[k_2]}\nabla\hat{\mathcal{H}}_\theta(u) - \hat{R}_\theta^{[k_3]}\nabla\hat{\mathcal{V}}_\theta(u) + k_4\hat{f}_\theta(u, x, t)\right),$$

where

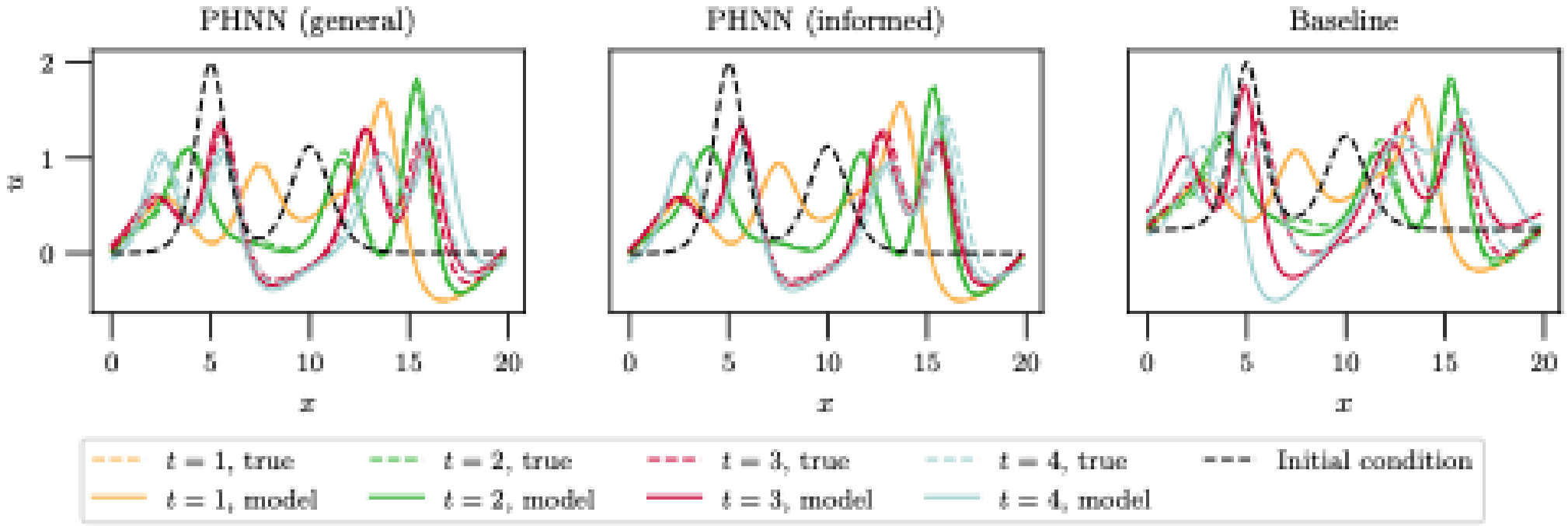
- $\hat{A}_\theta^{[k_1]}$, $\hat{S}_\theta^{[k_2]}$ and $\hat{R}_\theta^{[k_3]}$ are trainable **convolution operators** approximating A, S and R, where k_1, k_2, k_3 are the kernel sizes
- $\hat{\mathcal{H}}_\theta : \mathbb{R}^M \to \mathbb{R}$ and $\hat{\mathcal{V}}_\theta : \mathbb{R}^M \to \mathbb{R}$ are **convolutional NNs** with 3 layers and kernel sizes 2, 1, 1, and a summation $\mathbb{R}^M \to \mathbb{R}$ after the last layer
- $\hat{f}_\theta : \mathbb{R}^M \times \mathbb{R}^M \times \mathbb{R} \to \mathbb{R}^M$ is a **convolutional NN** with 3 layers and kernel sizes 1, 1, 1

PHNN for the forced KdV–Burgers equation



PHNN for the forced KdV–Burgers equation

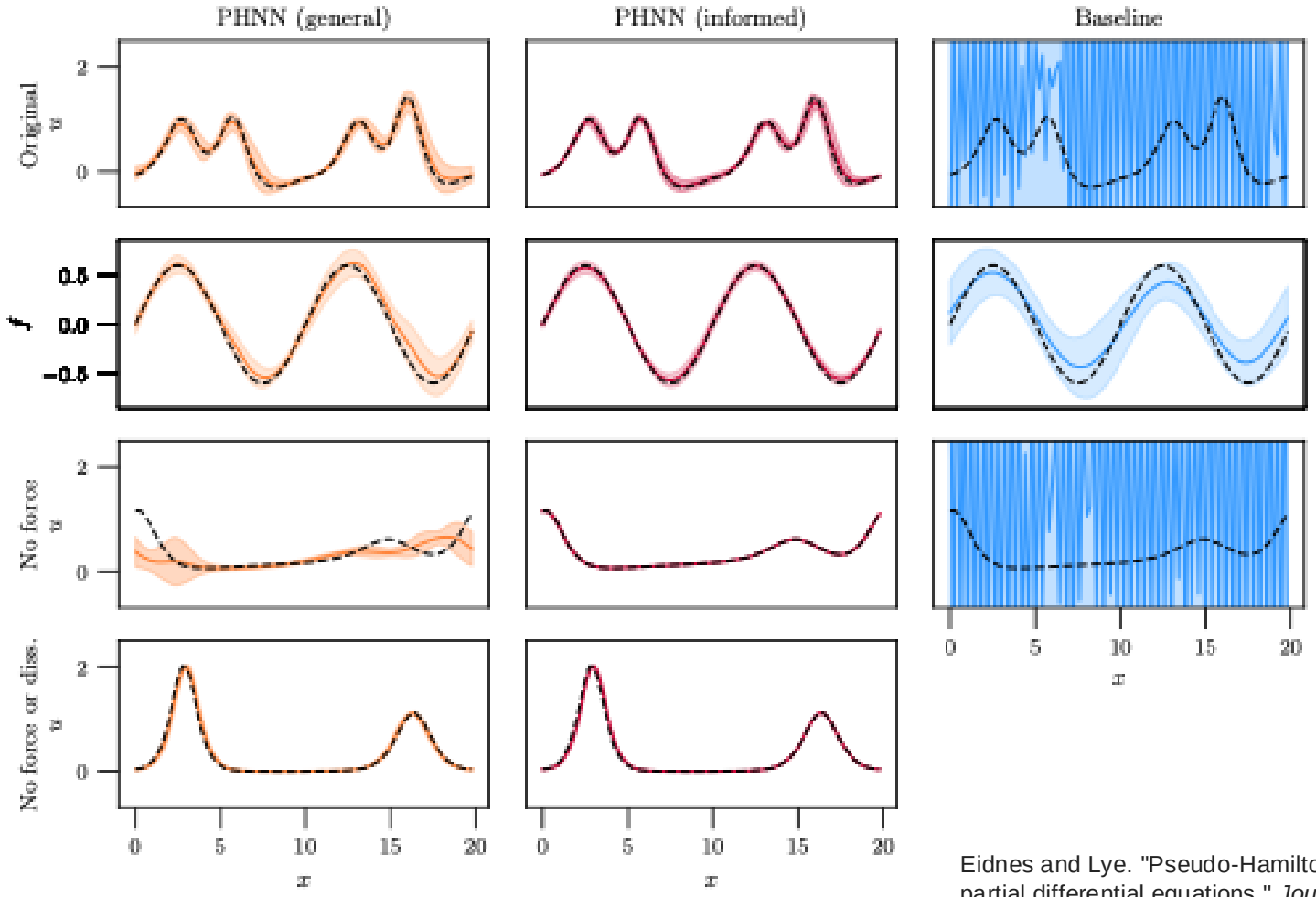
With $f(u, x, t) = \frac{3}{5} \sin\left(\frac{4\pi}{P} x\right)$:



Training data: 60 states, obtained by integrating from 10 initial states and evaluating at every $\Delta t = 0.1$ until $t = 0.5$.

Eidnes and Lye. "Pseudo-Hamiltonian neural networks for learning partial differential equations." *Journal of Computational Physics* 500 (2024): 112738.

PHNN for the forced KdV–Burgers equation



Eidnes and Lye. "Pseudo-Hamiltonian neural networks for learning partial differential equations." *Journal of Computational Physics* 500 (2024): 112738.

Pseudo-Hamiltonian PDEs

$$u_t = S(u^J, x)\frac{\delta \mathcal{H}}{\delta u}[u] - R(u^J, x)\frac{\delta \mathcal{V}}{\delta u}[u] + f(u^J, x, t)$$

Example: The forced KdV–Burgers (or viscous KdV) equation

$$u_t + \eta u u_x + \gamma^2 u_{xxx} - \nu u_{xx} = 0 \quad (u^J, x, t),$$

where

$$S(u^J, x) = \frac{\partial}{\partial x}, \quad \mathcal{H} = \int_{\mathbb{R}} \left(-\frac{\eta}{6} u^3 + \frac{\gamma}{2} u_x^2 \right) dx$$

and

$$R(u^J, x) = I, \quad \mathcal{V} = \frac{\nu}{2} \int_{\mathbb{R}} u_x^2 \, dx.$$

Pseudo-Hamiltonian PDEs

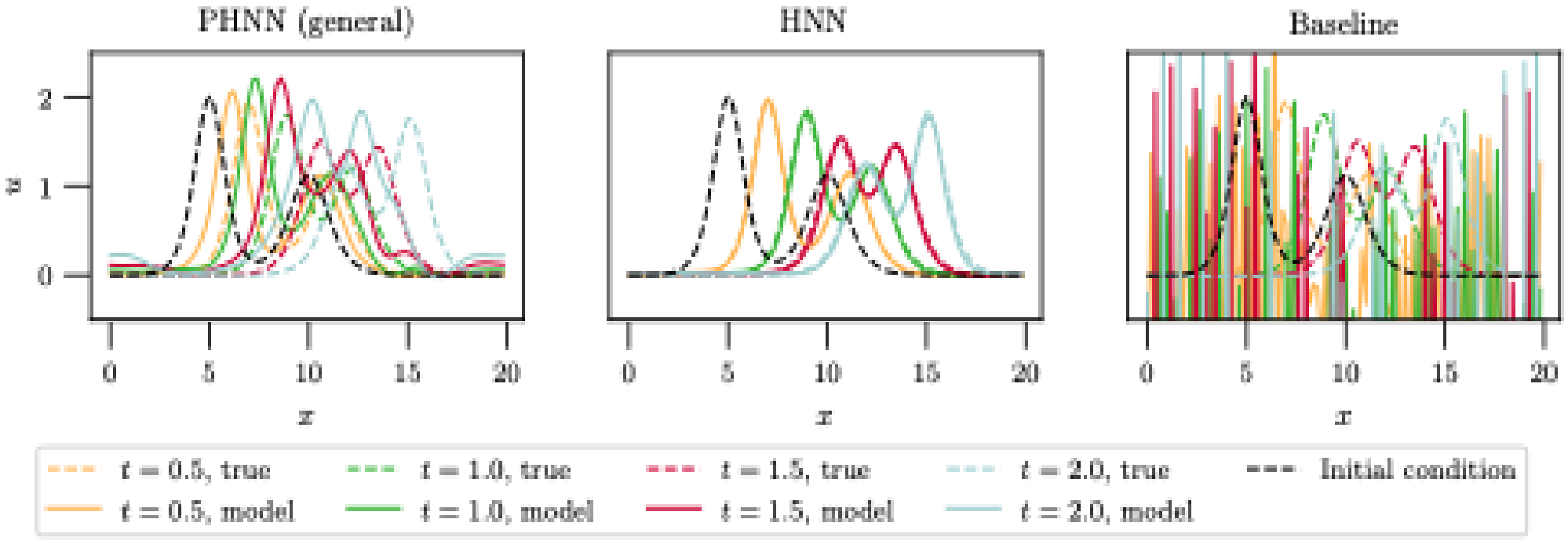
The full PHNN model for PDEs is given by

$$\hat{g}_\theta(u, x, t) = (\hat{A}_\theta^{[k_1]})^{-1}\left(\hat{S}_\theta^{[k_2]}\nabla\hat{\mathcal{H}}_\theta(u) - \hat{R}_\theta^{[k_3]}\nabla\hat{\mathcal{V}}_\theta(u) + k_4\hat{f}_\theta(u, x, t)\right),$$

where

- $\hat{A}_\theta^{[k_1]}$, $\hat{S}_\theta^{[k_2]}$ and $\hat{R}_\theta^{[k_3]}$ are trainable **convolution operators** approximating A, S and R, where k_1, k_2, k_3 are the kernel sizes
- $\hat{\mathcal{H}}_\theta : \mathbb{R}^M \to \mathbb{R}$ and $\hat{\mathcal{V}}_\theta : \mathbb{R}^M \to \mathbb{R}$ are **convolutional NNs** with 3 layers and kernel sizes 2, 1, 1, and a summation $\mathbb{R}^M \to \mathbb{R}$ after the last layer
- $\hat{f}_\theta : \mathbb{R}^M \times \mathbb{R}^M \times \mathbb{R} \to \mathbb{R}^M$ is a **convolutional NN** with 3 layers and kernel sizes 1, 1, 1

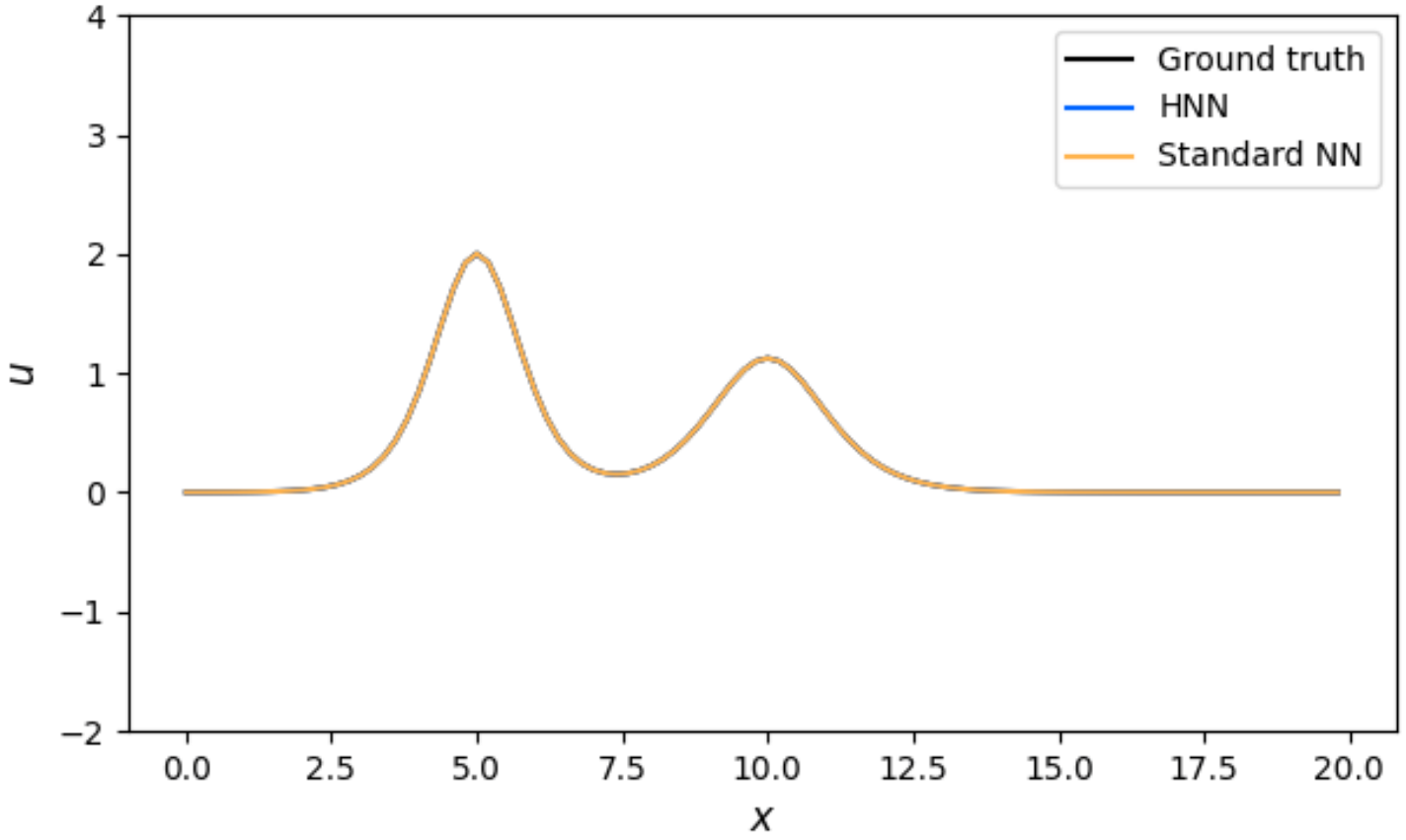
HNN for the KdV equation



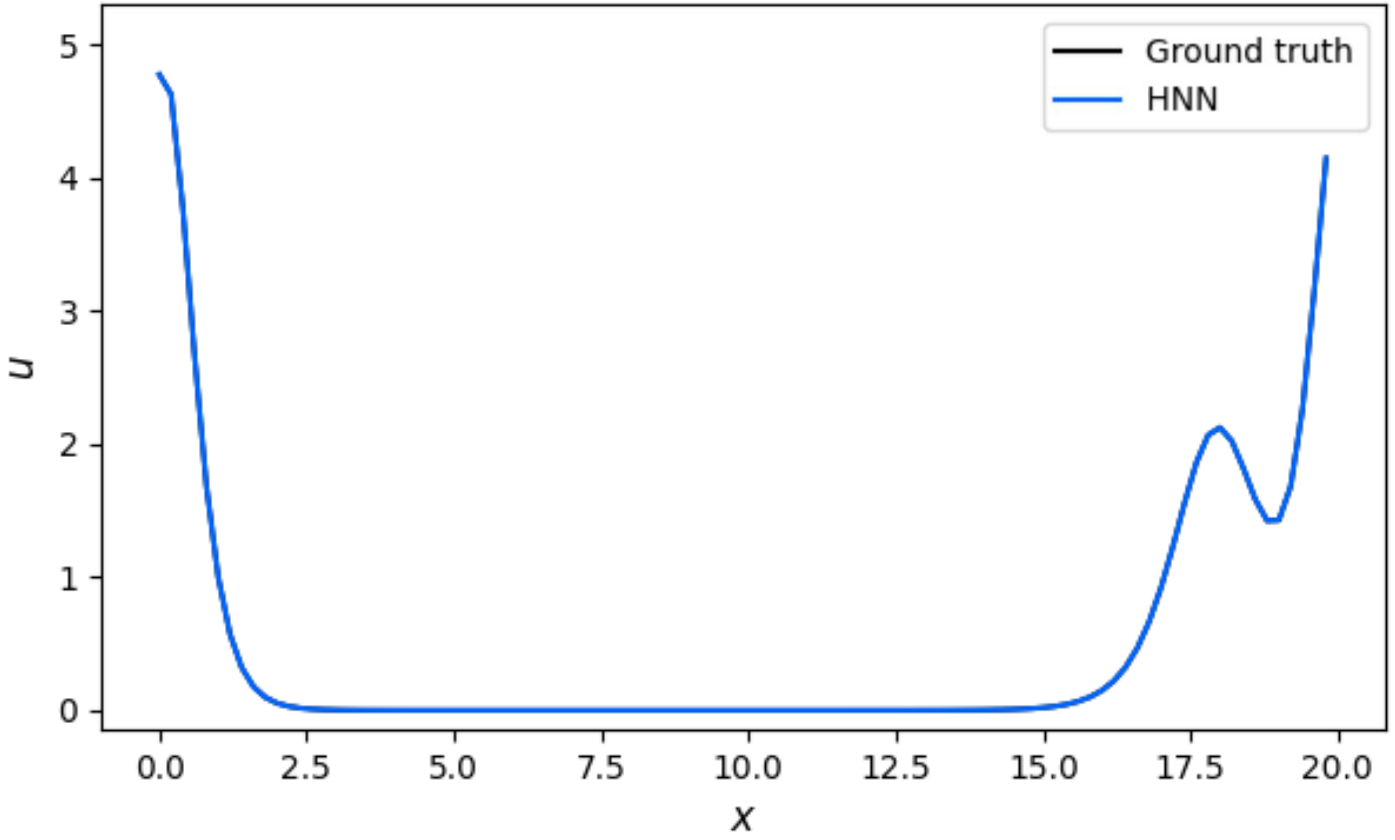
Training data: 420 states, obtained by integrating from 20 initial states and evaluating at every $\Delta t = 0.01$ until $t = 0.2$.

Eidnes and Lye. "Pseudo-Hamiltonian neural networks for learning partial differential equations." *Journal of Computational Physics* 500 (2024): 112738.

HNN for the KdV equation



HNN for the KdV equation



Exercise 2: HNN for the KdV equation

Set up and train an HNN model to learn the KdV equation

- Use a convolutional layer to learn the finite difference approximation of spatial derivatives
  - What should the kernel size be?
- Use padding to ensure periodic boundaries

```python
class HNN(nn.Module):
    def __init__(self, conv_kernel_size=SET_KERNEL_SIZE):
        super().__init__()
        self.padding_size = 1
        self.hamiltonian_net = nn.Sequential(
            nn.Conv1d(1, 100, kernel_size=conv_kernel_size),
            nn.Tanh(),
            nn.Conv1d(100, 100, kernel_size=1),
            nn.Tanh(),
            nn.Conv1d(100, 100, kernel_size=1, bias=None)
        )

    def forward_padding(self, x):
        return torch.cat([x, x[..., :self.padding_size]], dim=-1)

    def summation(self, x):
        return x.sum(dim=tuple(range(1, x.ndim)), keepdim=True)

    def hamiltonian(self, u):
        u_padded = self.forward_padding(u)
        H = self.hamiltonian_net(u_padded)
        return self.summation(H)

    def forward(self, u, dx):
        H = self.hamiltonian(u)
        dH = torch.autograd.grad(H.sum(), u, create_graph=True)[0]
        dH_padded = torch.cat([dH[..., u.shape[-1] - 1 :], dH, dH[..., :1]], dim=-1)
        S = (torch.tensor([-1., 0., 1.], dtype=torch.float32) / (2 * dx)).reshape(1, 1, 3).to(u.device)
        return torch.nn.functional.conv1d(dH_padded, S)
```

SINTEF

Technology for a better society