Machine Learning techniques for numerical solvers

Day 2: supervised learning

Maria Han Veiga
Mini-course SUSTech
11.03 - 14.03

The Ohio State University
College of Arts and Sciences

Schedule

- **Monday:** Introduction to Machine Learning (1h) 17:00-18:00
- **Tuesday:** Computational Framework + Supervised learning: integrating data-driven methods within a numerical solver + Hands-on session (2h) 14:00-16:00
- **Wednesday:** Unsupervised learning: Physics informed neural networks (1h) 11:00-12:00
- **Thursday:** Reinforcement Learning ? (1h) 11:00-12:00

Day 3: Unsupervised learning

Outline

- Physics informed neural networks (PINNs)
  - Raissi et al. 2017
  - PINNs now

- Hands-on session:
  - Viscous Burgers equation in 1D
  - ODE — damped oscillator

Motivation

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x,t)$ denotes the solution and $\mathscr{L}(\,\cdot\,;\lambda)$ is a nonlinear operator parametrised by λ.

Motivation

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x,t)$ denotes the solution and $\mathscr{L}(\,\cdot\,;\lambda)$ is a nonlinear operator parametrised by λ.

- **Idea:** Use a data-driven method to solve the PDE problem.
    - Dissanayake et al, 1994, Lagalis et al. 1998 — Artificial neural networks to solve ODEs/PDEs
    - Raissi et al. 2017, Sirignano et al. 2017 — Novel minimisation problem and modern deep learning techniques

Physics informed neural networks (PINNs)

- According to authors:
- Neural networks trained to "*solve supervised learning tasks while respecting physical laws (PDEs)*" (Raissi et al. 2017)
  - Data-driven solution
- Two types of algorithms: (Raissi et al. 2019)
  - Family of data-efficient patio-temporal function approximators
  - Arbitrary accurate RK time steppers

Physics informed neural networks (PINNs)

- According to authors:
- Neural networks trained to *“solve supervised learning tasks while respecting physical laws (PDEs)”* (Raissi et al. 2017)
  - Data-driven solution
- Two types of algorithms: (Raissi et al. 2019)
  - Family of data-efficient patio-temporal function approximators
  - Arbitrary accurate RK time steppers
- I call these unsupervised learning, or maybe self-supervised learning, because we don’t have explicitly a dataset $S = \{(x_i, y_i), i = 1,\ldots,m\}$.

Setup

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u; \lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x, t)$ denotes the solution and $\mathscr{L}(\,\cdot\,; \lambda)$ is a nonlinear operator parametrised by λ.

Setup

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x,t)$ denotes the solution and $\mathscr{L}(\,\cdot\,;\lambda)$ is a nonlinear operator parametrised by λ.

e.g. Viscous Burger's equation in 2D:

$$\mathscr{L}(u;\lambda) = \lambda_1 u \partial_x u - \lambda_2 u \partial_{xx} u$$

Setup

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u; \lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x, t)$ denotes the solution and $\mathscr{L}(\,\cdot\,; \lambda)$ is a nonlinear operator parametrised by λ.

Task:

1) Given λ, what is $u(x, t)$ that fulfils the PDE? (Data-driven solution of PDEs)

2) Find λ that best describes observations $u(x_i, t_i)$. (Data-driven discovery of PDEs)

Setup

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

where $u(x,t)$ denotes the solution and $\mathscr{L}(\,\cdot\,;\lambda)$ is a nonlinear operator parametrised by λ.

Task:

1) Given λ, what is $u(x,t)$ that fulfils the PDE? (Data-driven solution of PDEs)

2) Find λ that best describes observations $u(x_i, t_i)$. (Data-driven discovery of PDEs)

Data-driven solutions

$$
\begin{aligned}
\partial_t u + \mathscr{L}(u;\lambda) &= 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T] \\
u(x, t = 0) &= u_0(x) \\
u(\partial\Omega, t) &= g(t)
\end{aligned}
$$

Data-driven solutions

$$
\begin{aligned}
\partial_t u + \mathscr{L}(u;\lambda) &= 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T] \\
u(x, t=0) &= u_0(x) \\
u(\partial\Omega, t) &= g(t)
\end{aligned}
$$

- Let the solution of the PDE be given by a neural network $u(t,x) = \mathcal{N}\mathcal{N}(t,x;\theta)$

Data-driven solutions

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$
$$u(x, t = 0) = u_0(x)$$
$$u(\partial\Omega, t) = g(t)$$

- Let the solution of the PDE be given by a neural network $u(t, x) = \mathcal{N}\mathcal{N}(t, x; \theta)$
- To solve the PDE, we want the residual $f(u; t, x) := \partial_t u + \mathscr{L}(u)$ to be zero at all (x, t) and that the neural network fulfils the initial and boundary data.

Data-driven solutions

$$
\begin{aligned}
\partial_t u + \mathscr{L}(u;\lambda) &= 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T] \\
u(x, t=0) &= u_0(x) \\
u(\partial\Omega, t) &= g(t)
\end{aligned}
$$

- Let the solution of the PDE be given by a neural network $u(t,x) = \mathcal{N}\mathcal{N}(t,x;\theta)$
- To solve the PDE, we want the residual $f(u;t,x) := \partial_t u + \mathscr{L}(u)$ to be zero at all (x,t) and that the neural network fulfils the initial and boundary data.
- Introduce loss:

$$
\begin{aligned}
L(\mathcal{N}\mathcal{N}) = \frac{1}{N_i}\lambda_i \sum_{i=1}^{N_i} |\mathcal{N}\mathcal{N}(x_i,0) - u(x_i,0)|^2 &+ \frac{1}{N_b}\lambda_b \sum_{i=1}^{N_b} |\mathcal{N}\mathcal{N}(\partial\Omega_i, t_i) - g(x_i)|^2 \\
&+ \frac{1}{N_f}\lambda_f \sum_{i=1}^{N_f} |f(x_i, t_i)|^2
\end{aligned}
$$

Data-driven solutions

$$\begin{aligned} \partial_t u + \mathscr{L}(u;\lambda) &= 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T] \\ u(x, t = 0) &= u_0(x) \\ u(\partial\Omega, t) &= g(t) \end{aligned}$$

- Let the solution of the PDE be given by a neural network $u(t, x) = \mathcal{N}\mathcal{N}(t, x; \theta)$
- To solve the PDE, we want the residual $f(u; t, x) := \partial_t u + \mathscr{L}(u)$ to be zero at all (x, t) and that the neural network fulfils the initial and boundary data.
- Introduce loss:

$$L(\mathcal{N}\mathcal{N}) = \boxed{\frac{1}{N_i}\lambda_i \sum_{i=1}^{N_i} |\mathcal{N}\mathcal{N}(x_i,0) - u(x_i,0)|^2 + \frac{1}{N_b}\lambda_b \sum_{i=1}^{N_b} |\mathcal{N}\mathcal{N}(\partial\Omega_i, t_i) - g(x_i)|^2} + \frac{1}{N_f}\lambda_f \sum_{i=1}^{N_f} |f(x_i, t_i)|^2$$

Data-driven solutions

$$\partial_t u + \mathscr{L}(u;\lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$
$$u(x, t=0) = u_0(x)$$
$$u(\partial\Omega, t) = g(t)$$

- Let the solution of the PDE be given by a neural network $u(t,x) = \mathcal{N}\mathcal{N}(t,x;\theta)$
- To solve the PDE, we want the residual $f(u;t,x) := \partial_t u + \mathscr{L}(u)$ to be zero at all (x,t) and that the neural network fulfils the initial and boundary data.
- Introduce loss:

$$L(\mathcal{N}\mathcal{N}) = \frac{1}{N_i}\lambda_i \sum_{i=1}^{N_i} |\mathcal{N}\mathcal{N}(x_i,0) - u(x_i,0)|^2 + \frac{1}{N_b}\lambda_b \sum_{i=1}^{N_b} |\mathcal{N}\mathcal{N}(\partial\Omega_i, t_i) - g(x_i)|^2$$
$$\boxed{+\frac{1}{N_f}\lambda_f \sum_{i=1}^{N_f} |f(x_i, t_i)|^2}$$

Data-driven solutions

$$L(\mathcal{N}\mathcal{N}) = \frac{1}{N_i}\lambda_i \sum_{i=1}^{N_i} |\,\mathcal{N}\mathcal{N}(x_i,0) - u(x_i,0)\,|^2 + \frac{1}{N_b}\lambda_b \sum_{i=1}^{N_b} |\,\mathcal{N}\mathcal{N}(\partial\Omega_i, t_i) - g(x_i)\,|^2$$

$$+ \frac{1}{N_f}\lambda_f \sum_{i=1}^{N_f} |f(x_i, t_i)\,|^2$$

Data-driven solutions

$$L(\mathcal{N}\mathcal{N}) = \frac{1}{N_i}\lambda_i \sum_{i=1}^{N_i} |\mathcal{N}\mathcal{N}(x_i, 0) - u(x_i, 0)|^2 + \frac{1}{N_b}\lambda_b \sum_{i=1}^{N_b} |\mathcal{N}\mathcal{N}(\partial\Omega_i, t_i) - g(x_i)|^2$$

$$+ \frac{1}{N_f}\lambda_f \sum_{i=1}^{N_f} |f(x_i, t_i)|^2$$

The loss is composed by two types of nodes:

- Initial and boundary nodes
- Collocation points

Data-driven solutions

- No theoretical guarantees but if PDE is well-posed, seems like optimiser will find a solution.

Data-driven solutions

- No theoretical guarantees but if PDE is well-posed, seems like optimiser will find a solution.
- Example from PINNs paper (Raissi et al 2017), solve the 1D Burgers' equation with Dirichlet boundary conditions

$$\partial_t u + u\partial_x u - (0.01/\pi)\partial_{xx} u = 0 \quad x \in [-1,1], \quad t \in [0,1]$$

$$u(x,0) = -\sin(\pi x)$$

$$u(-1,t) = u(1,t) = 0$$

Data-driven solutions

- No theoretical guarantees but if PDE is well-posed, seems like optimiser will find a solution.
- Example from PINNs paper (Raissi et al 2017), solve the 1D Burgers' equation with Dirichlet boundary conditions

$$\partial_t u + u\partial_x u - (0.01/\pi)\partial_{xx}u = 0 \quad x \in [-1,1], \quad t \in [0,1]$$
$$u(x,0) = -\sin(\pi x)$$
$$u(-1,t) = u(1,t) = 0$$

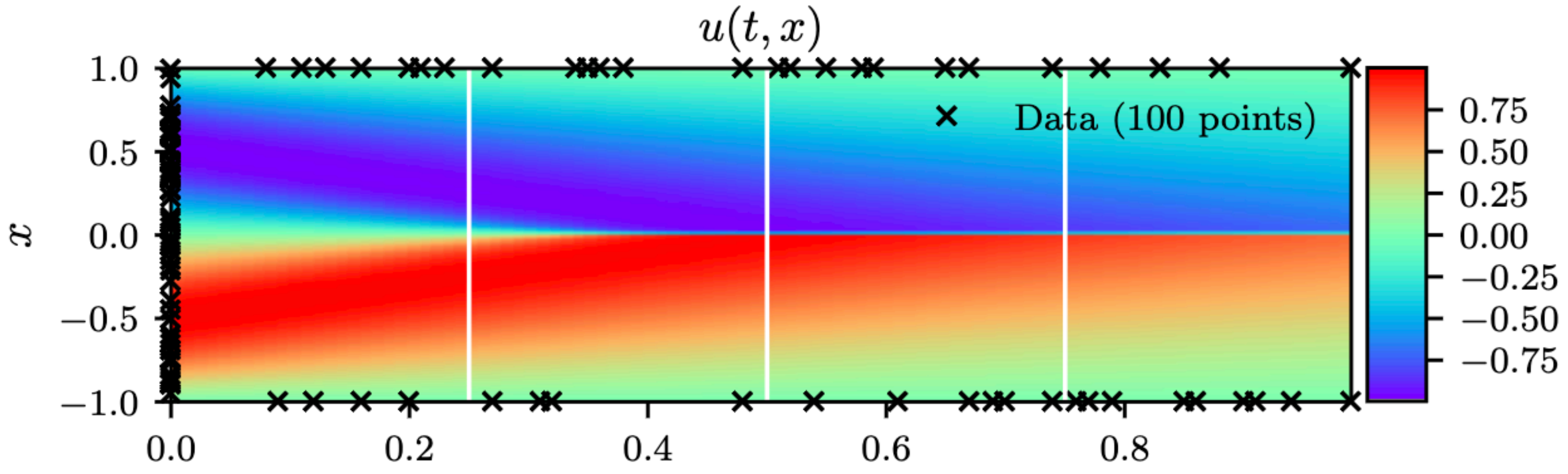
Define $f := \partial_t u + u\partial_x u - (0.01/\pi)\partial_{xx}u$.

Let $u(x, t) = \mathcal{N}\mathcal{N}(x, t)$ and minimise Physics Informed Loss:

$$L(\mathcal{N}\mathcal{N}) = \text{Initial data loss} + \text{Boundary data loss} + \text{Residual loss}$$

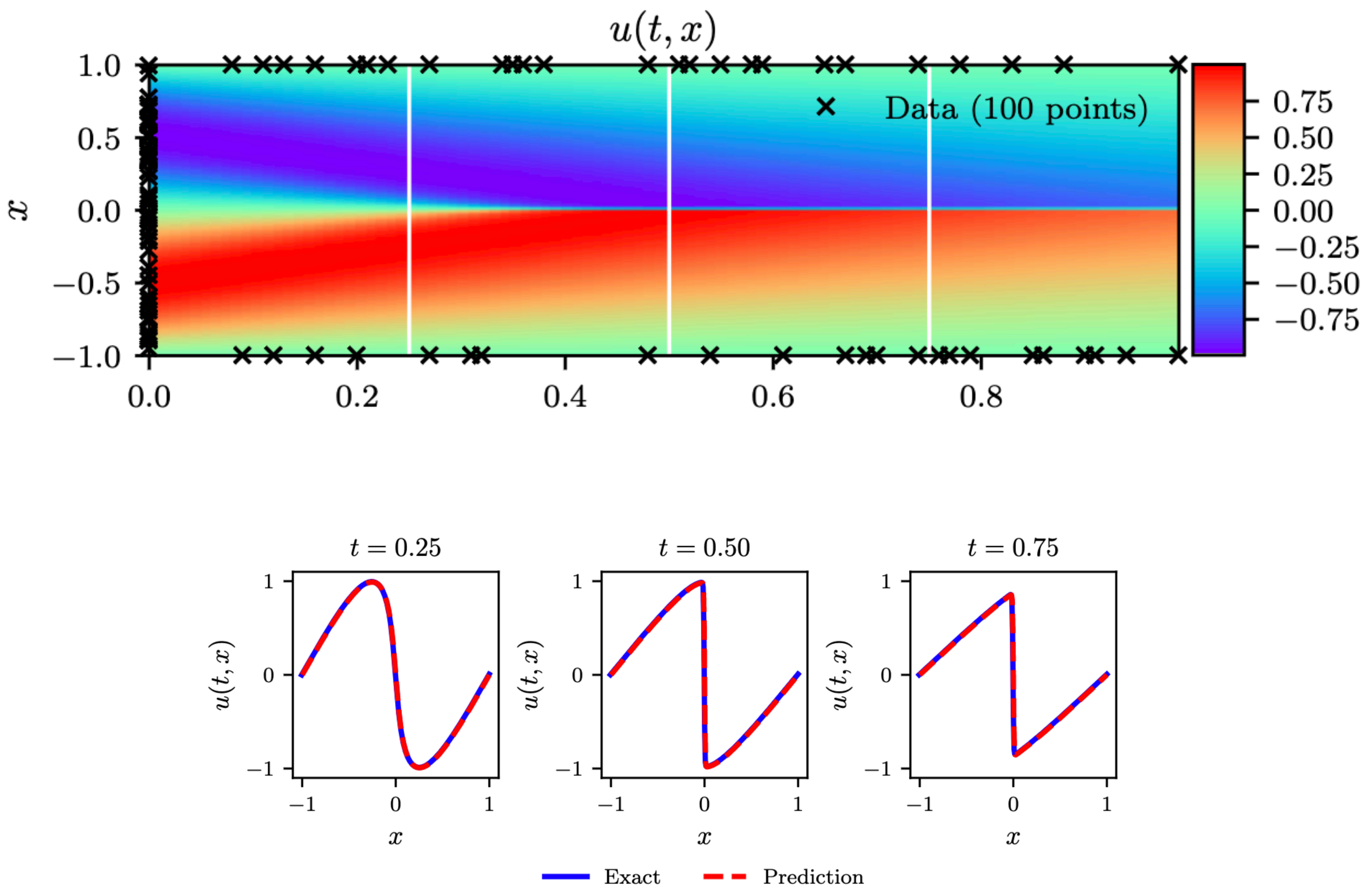
Data-driven solutions

- Example from PINNs paper (Raissi et al 2017)



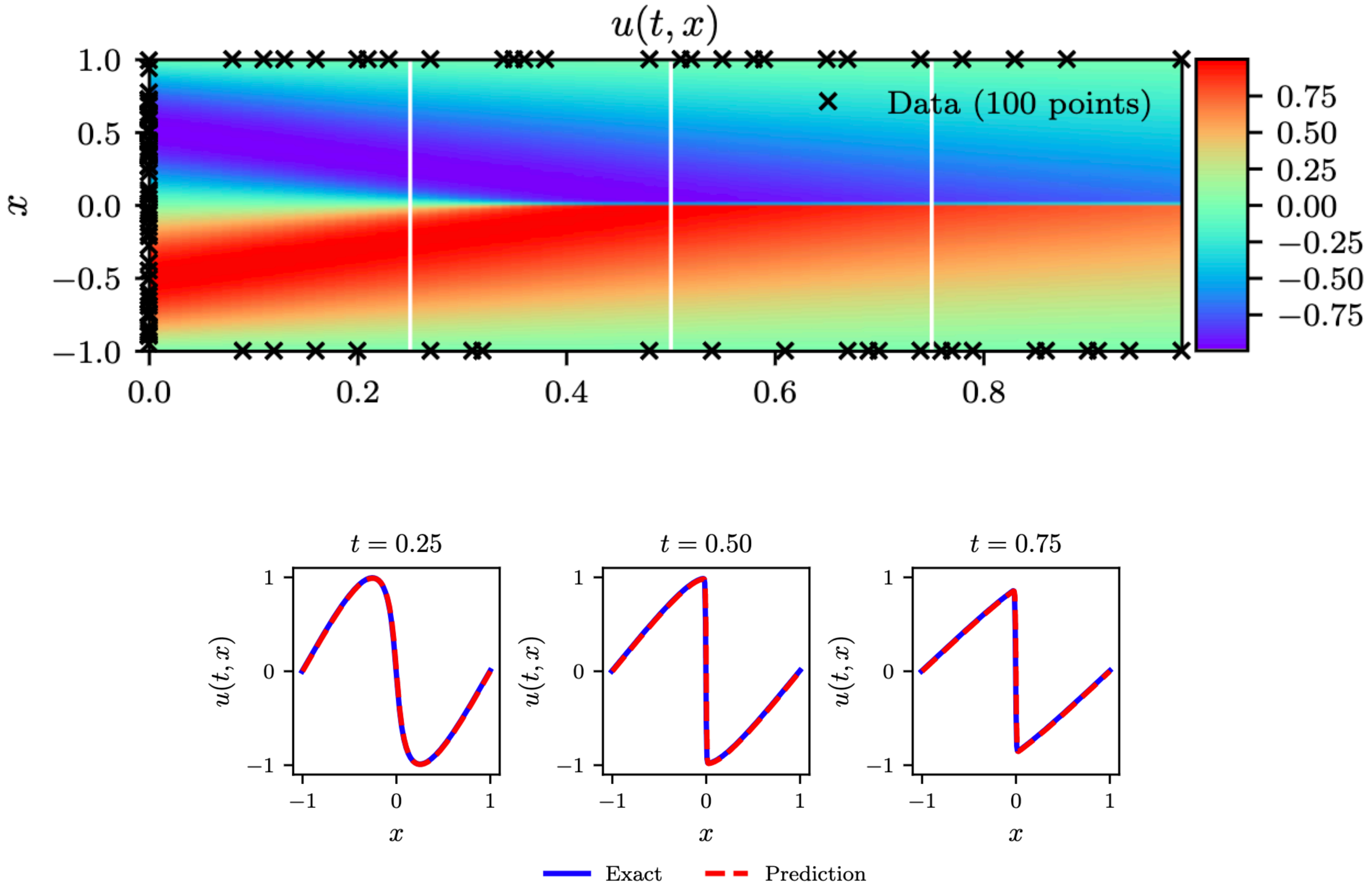
Data-driven solutions

- Example from PINNs paper (Raissi et al 2017)



Data-driven solutions

- Example from PINNs paper (Raissi et al 2017)



- Training details:
  - 100 points for boundary and initial data
  - 10000 colocation points (for residual), sampled using Latin Hypercube Sampling strategy. (2D space)

Data-driven solutions

- In original PINNs paper (Raissi et al 2017)
  - Shrödinger equation in 1D (complex numbers)
  - Allen-Cahn Equation

Pros and cons

Pros

- Mesh free
- Mostly unsupervised
- Can work with noisy data

Cons:

- Convergence properties are not well understood
- Computational (training) cost can be much higher than a traditional solver
- Poor scaling to large domains / high frequencies / more complex solutions

PINNs now

- Review paper (Cuomo et al 2022):
  - Different types of PINNs: beyond fully connected
    - CNNs, AutoEncoders, ResNet, Recurrent (often tailor made for different applications)
    - Different type of PDEs (see references in review paper):
      - Advection-diffusion-reaction problems
      - Diffusion problems
      - Advection problems
      - Navier-Stokes equations
      - Hyperbolic equations

https://link.springer.com/article/10.1007/s10915-022-01939-z

PINNs now

- Review paper (Cuomo et al 2022):
  - Different types of PINNs: beyond fully connected
    - CNNs, AutoEncoders, ResNet, Recurrent (often tailor made for different applications)
    - Different type of PDEs (see references in review paper):
      - Advection-diffusion-reaction problems
      - Diffusion problems
      - Advection problems
      - Navier-Stokes equations
      - **Hyperbolic equations**

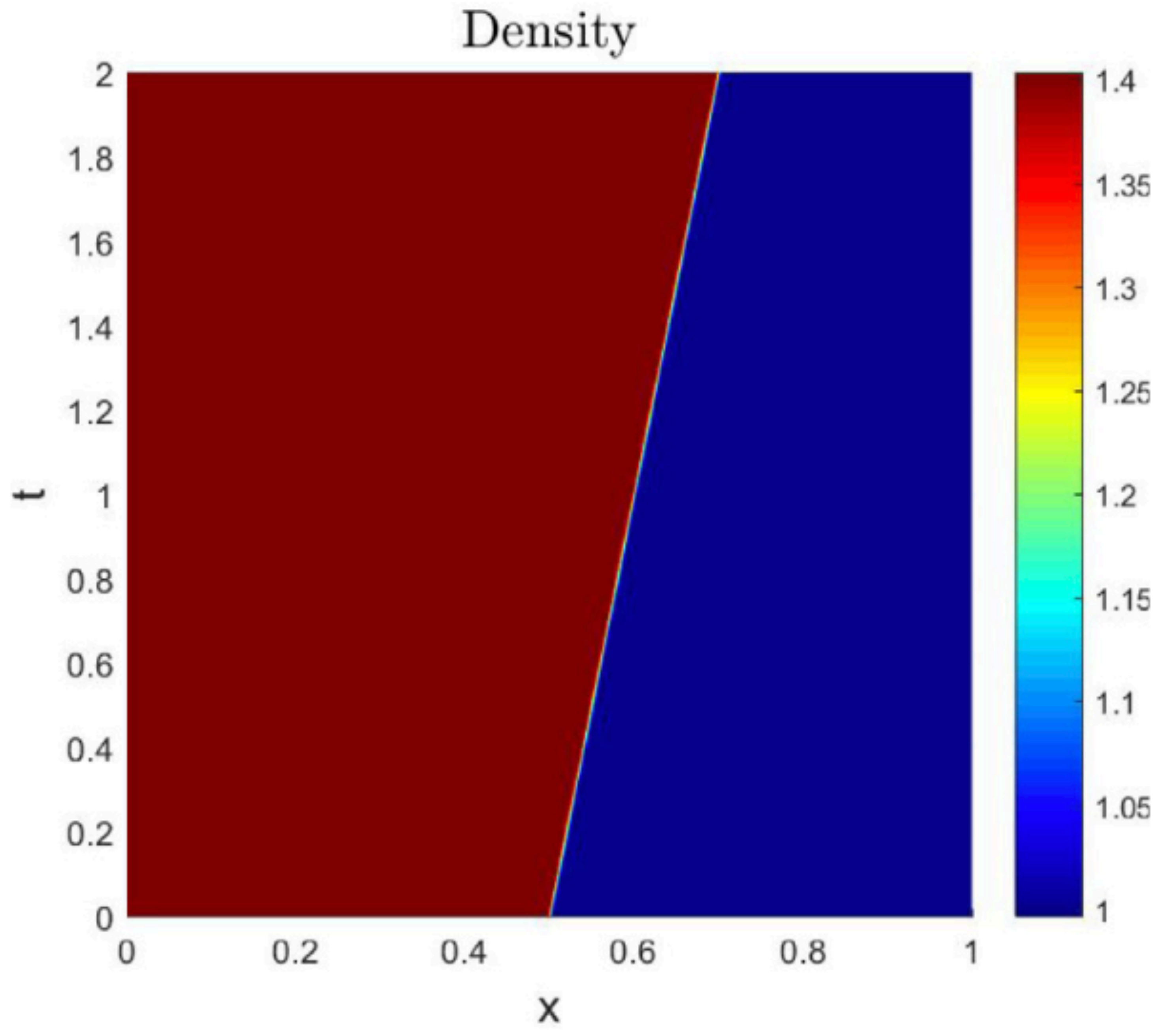
https://link.springer.com/article/10.1007/s10915-022-01939-z

PINNs now

- PINNs for Hyperbolic equations:
  - Mao et al. 2019: 1D Euler equation

PINNs now

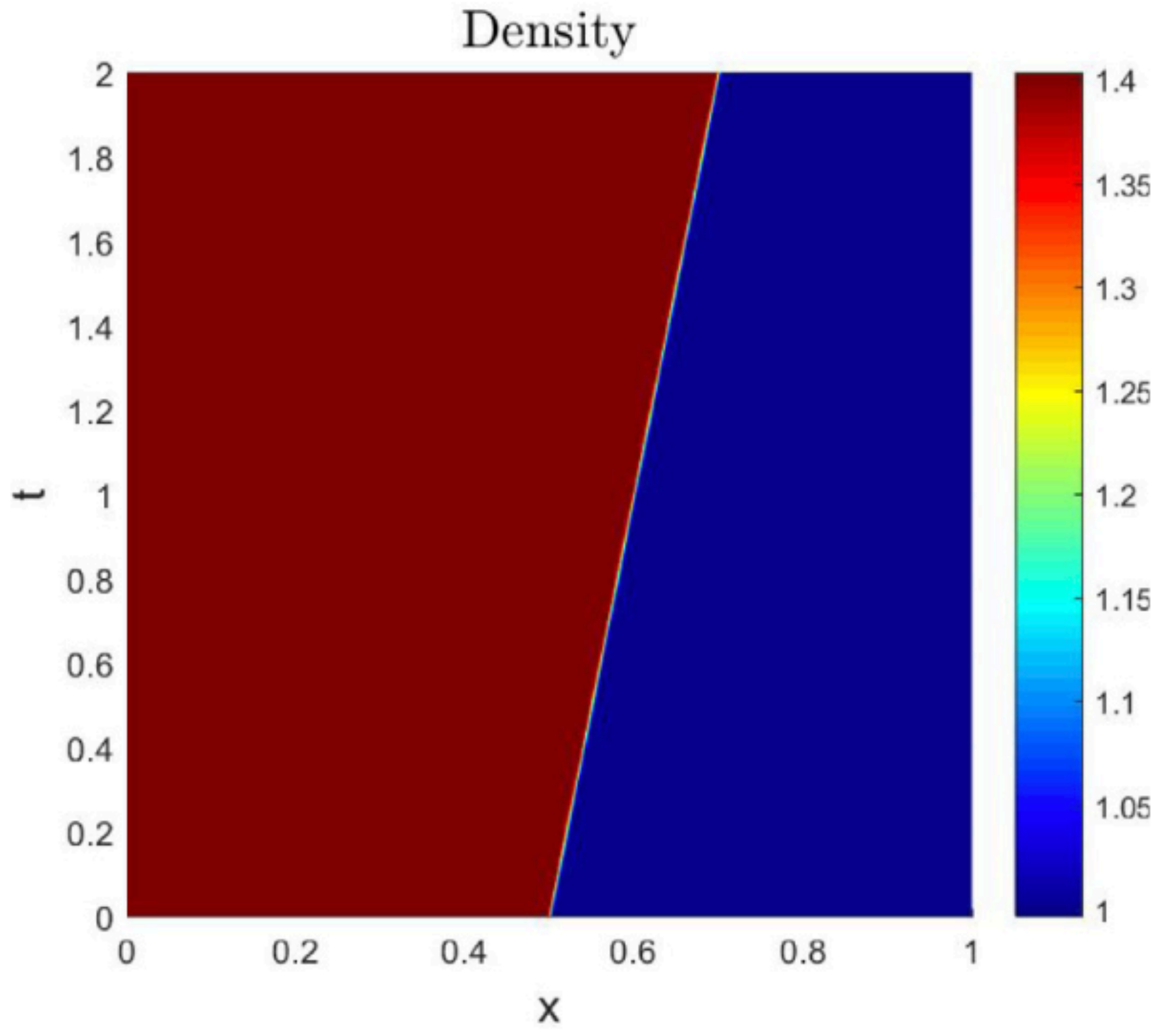
- PINNs for Hyperbolic equations:
  - Mao et al. 2019: 1D Euler equation
  - Example 1: Moving contact discontinuity, with shock position at $x = 0.5$



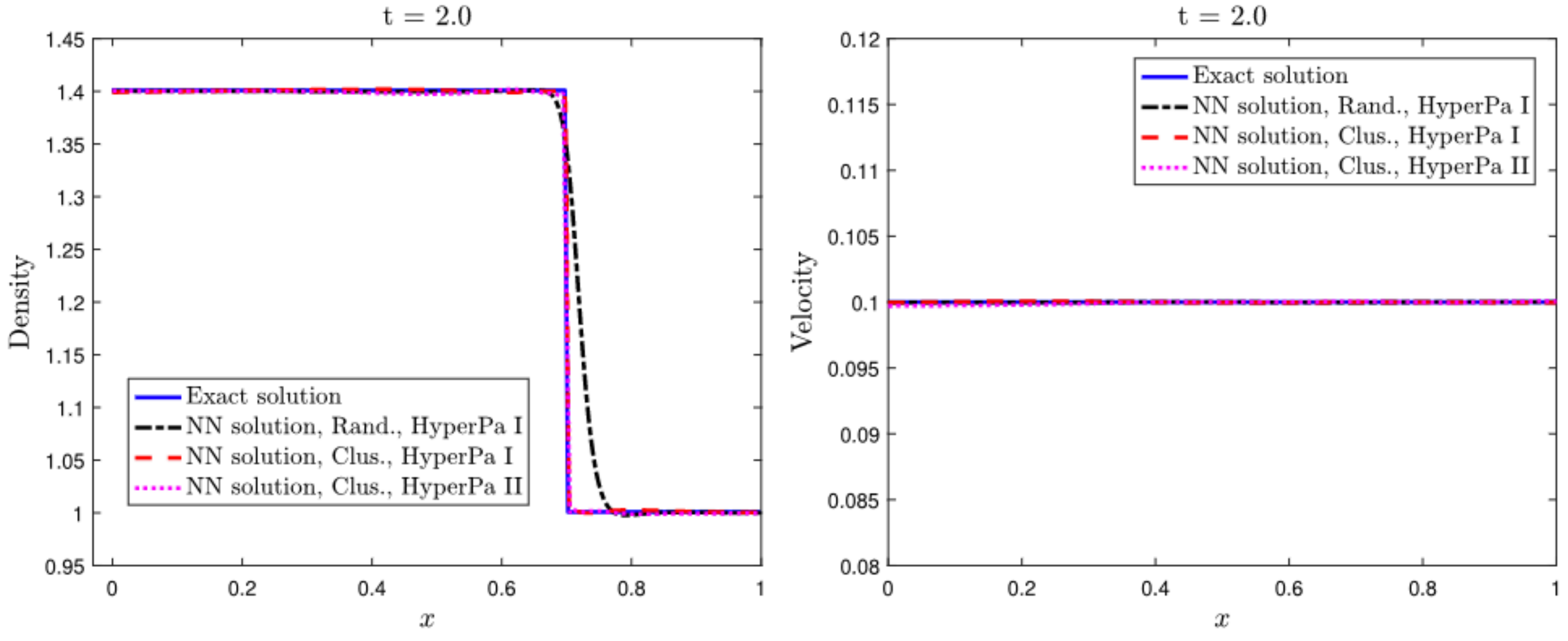
- Training nodes: 1120 points

PINNs now

- PINNs for Hyperbolic equations:
  - Mao et al. 2019: 1D Euler equation
  - Example 1: Moving contact discontinuity, with shock position at $x = 0.5$



- Training nodes: 1120 points



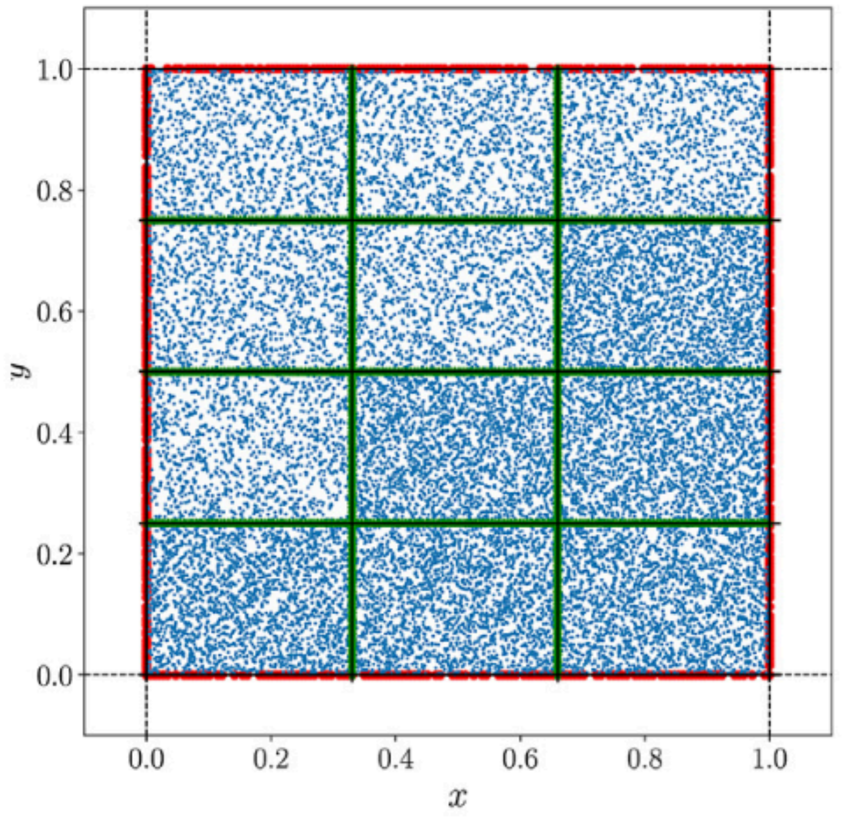
(a) (b)

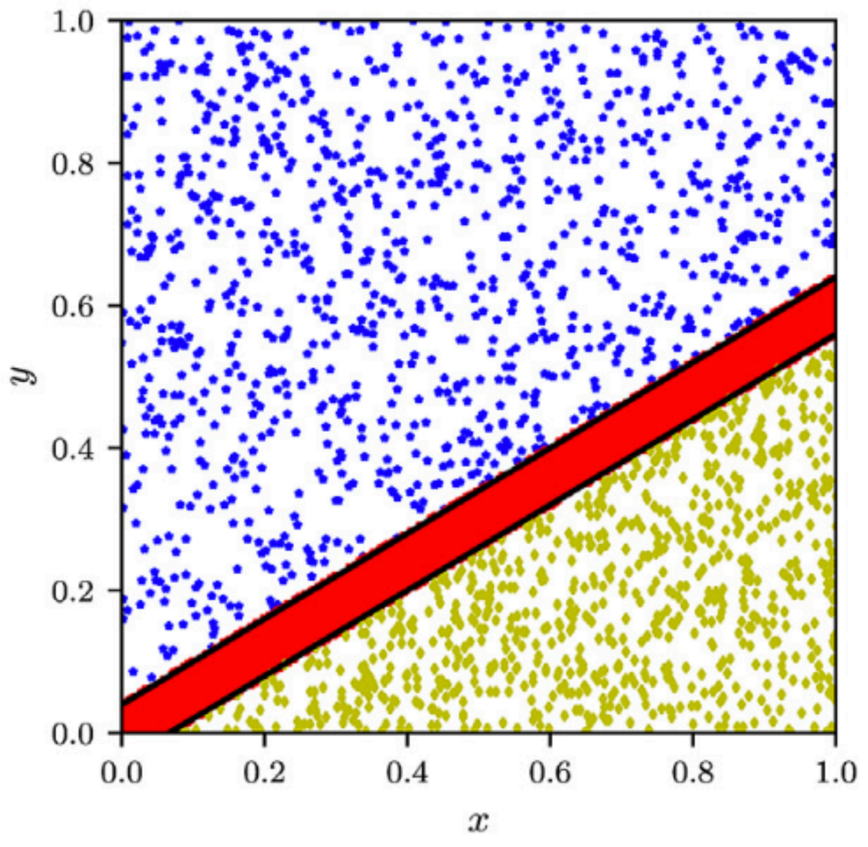
PINNs now

- PINNs for Hyperbolic equations:
    - Jagtap et al. 2020: 1D and 2D Euler equation
    - PINNs loss includes also conservation terms and domain decomposition
    - Example 2D Euler, IC: oblique shock wave problem

PINNs now

- PINNs for Hyperbolic equations:
  - Jagtap et al. 2020: 1D and 2D Euler equation
  - PINNs loss includes also conservation terms and domain decomposition
  - Example 2D Euler, IC: oblique shock wave problem

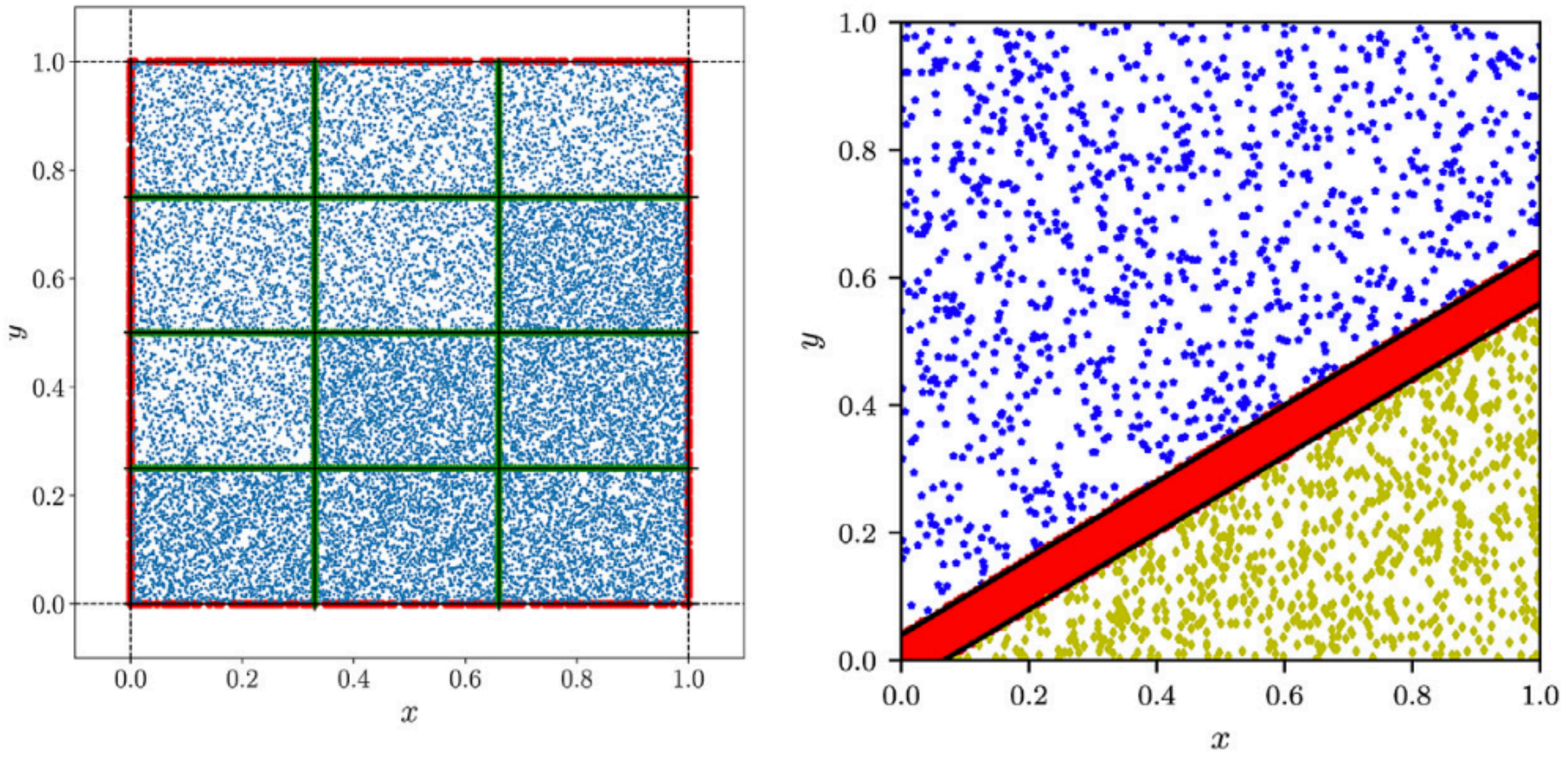




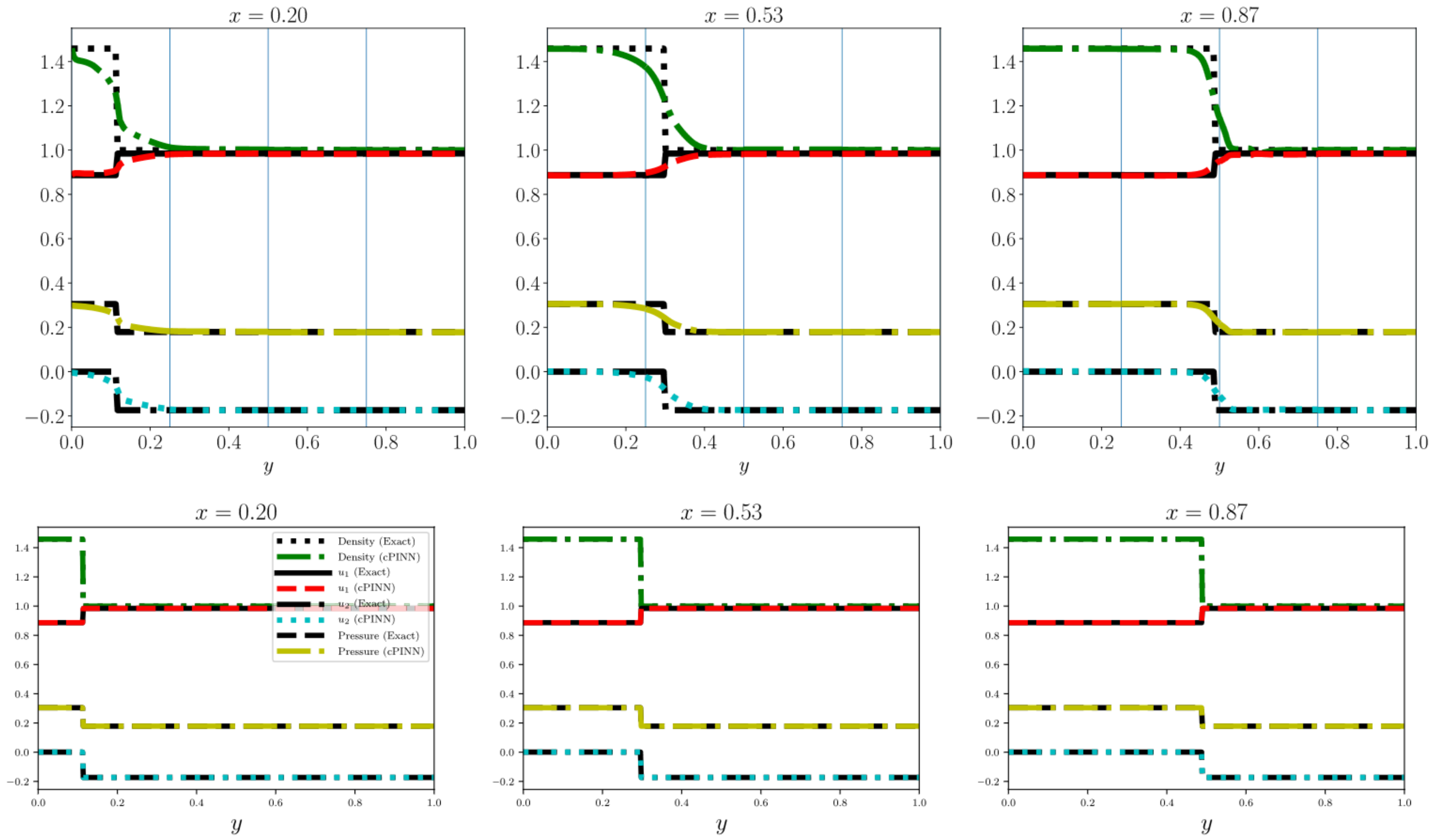
Domain decomposition

PINNs now

- PINNs for Hyperbolic equations:
  - Jagtap et al. 2020: 1D and 2D Euler equation
  - PINNs loss includes also conservation terms and domain decomposition
  - Example 2D Euler, IC: oblique shock wave problem



Domain decomposition



PINNs now

- Error analysis (Kutyniok, 2022).
- The global error between a trained deep neural network u_θ^* and the correct solution function u for a PDE can be bounded by:

$$\mathscr{R}(u_\theta^*) \leq \mathscr{E}_0 + 2\mathscr{E}_G + \mathscr{E}_A$$

PINNs now

- Error analysis (Kutyniok, 2022).
- The global error between a trained deep neural network u_θ^* and the correct solution function u for a PDE can be bounded by:

$$\mathscr{R}(u_\theta^*) \leq \mathscr{E}_0 + 2\mathscr{E}_G + \mathscr{E}_A$$

- $\mathscr{E}_0$ is the *optimisation error*, accounting for GD not finding the absolute minimum of the loss

PINNs now

- Error analysis (Kutyniok, 2022).
- The global error between a trained deep neural network u_θ^* and the correct solution function u for a PDE can be bounded by:

$$\mathcal{R}(u_\theta^*) \leq \mathcal{E}_0 + 2\mathcal{E}_G + \mathcal{E}_A$$

- $\mathcal{E}_0$ is the *optimisation error*, accounting for GD not finding the absolute minimum of the loss
- $\mathcal{E}_g$ is the *generalisation gap* (Mishra et al. 2022)

PINNs now

- Error analysis (Kutyniok, 2022).
- The global error between a trained deep neural network u_θ^* and the correct solution function u for a PDE can be bounded by:

$$\mathscr{R}(u_\theta^*) \leq \mathscr{E}_0 + 2\mathscr{E}_G + \mathscr{E}_A$$

- $\mathscr{E}_0$ is the *optimisation error*, accounting for GD not finding the absolute minimum of the loss
- $\mathscr{E}_g$ is the *generalisation gap* (Mishra et al. 2022)
- $\mathscr{E}_A$ is the *approximation error*, the ability of the neural network to approximate the exact solution (De Ryck et al. 2021)

PINNs inverse problems

- So far we focused on solving the *forward problem,* however, it seems like PINNs is quite effective to solve inverse problems / data-driven discovery of PDEs.

PINNs inverse problems

- So far we focused on solving the *forward problem,* however, it seems like PINNs is quite effective to solve inverse problems / data-driven discovery of PDEs.

- Parametrised, nonlinear PDE(s)

$$\partial_t u + \mathscr{L}(u; \lambda) = 0, \quad x \in \Omega \subset \mathbb{R}^D, \quad t \in [0,T]$$

  - where $u(x, t)$ denotes the solution and $\mathscr{L}(\,\cdot\,;\lambda)$ is a nonlinear operator parametrised by λ.

- One has observations of the solution $u(x, t)$, find the correct λ
- $\mathscr{L}(u, \cdot\,) := (\,\cdot\,)\partial_x u + (\,\cdot\,)\partial_{xx} u + (\,\cdot\,)\partial_y u + (\,\cdot\,)\partial_{yy} u + \ldots$

Hands-on

PINNs for 1D viscous burgers and ODE

PDE

- Solve the 1D Burgers' equation with Dirichlet boundary conditions

$$\partial_t u + u\partial_x u - (0.01/\pi)\partial_{xx} u = 0 \quad x \in [-1,1], \quad t \in [0,1]$$

$$u(x,0) = -\sin(\pi x)$$

$$u(-1,t) = u(1,t) = 0$$

PDE

Define the sampling nodes

```
boundary_nodes = np.zeros([100,2])
boundary_nodes[0:50,0]=1
boundary_nodes[0:50,1]=np.random.uniform(low=0.0, high=1, size=(50,))
boundary_nodes[50:,0]=-1
boundary_nodes[50:,1]=np.random.uniform(low=0.0, high=1, size=(50,))
```

```
initial_data_nodes = np.zeros([100,2])
initial_data_nodes[:,0]=np.random.uniform(low=-1.0, high=1, size=(100,))
initial_data_nodes[:,1]=0
initial_data_values = -np.sin(np.pi*initial_data_nodes[:,0])
```

```
collocation_data_nodes = np.zeros([5000,2])
collocation_data_nodes[:,0]=np.random.uniform(low=-1.0, high=1, size=(5000,))
collocation_data_nodes[:,1]=np.random.uniform(low=0.0, high=1, size=(5000,))
```

PDE

Define the sampling nodes

```
boundary_nodes = np.zeros([100,2])
boundary_nodes[0:50,0]=1
boundary_nodes[0:50,1]=np.random.uniform(low=0.0, high=1, size=(50,))
boundary_nodes[50:,0]=-1
boundary_nodes[50:,1]=np.random.uniform(low=0.0, high=1, size=(50,))
```

```
initial_data_nodes = np.zeros([100,2])
initial_data_nodes[:,0]=np.random.uniform(low=-1.0, high=1, size=(100,))
initial_data_nodes[:,1]=0
initial_data_values = -np.sin(np.pi*initial_data_nodes[:,0])
```

$u(x, t = 0) = -sin(\pi x)$

```
collocation_data_nodes = np.zeros([5000,2])
collocation_data_nodes[:,0]=np.random.uniform(low=-1.0, high=1, size=(5000,))
collocation_data_nodes[:,1]=np.random.uniform(low=0.0, high=1, size=(5000,))
```

PDE

Build the neural network:

- 2-dimensional input (x, t), 1-dimensional output u
- 3 hidden layers
- 32 width in each hidden layer
- Activation: Tanh — important because it has to be differentiable!

PDE

Training cycle:

```
for i in range(5000):
    optimiser.zero_grad()

    lambda1, lambda2, lambda3 = 1,1e-4,1e-1

    # compute initial data loss
    u = pinn(initial_data_nodes_t)
    loss1 = torch.mean((torch.squeeze(u) - initial_data_values_t)**2)

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss2 = torch.mean((u - torch.zeros(u.shape))**2)

    # compute physics loss
    u = pinn(collocation_data_nodes_t)
    du = torch.autograd.grad(u, collocation_data_nodes_t,\
                             torch.ones([collocation_data_nodes_t.shape[0], 1]),\
                             retain_graph=True, create_graph=True)[0]
    du2 = torch.autograd.grad(du, collocation_data_nodes_t,\
                              torch.ones(collocation_data_nodes_t.shape),\
                              create_graph=True)[0]
    u_x  = du[:,[0]]
    u_t  = du[:,[1]]
    u_xx = du2[:,[0]]
    loss3 = torch.mean((u_t + u*u_x - .01 / np.pi *u_xx)**2)

    # backpropagate joint loss, take optimiser step
    loss = lambda1*loss1 + lambda2*loss2 + lambda1*loss3
    loss.backward()
    optimiser.step()
```

PDE

Training cycle:

$$|\mathcal{N}\mathcal{N}(x_i) - u_0(x_i)| \longrightarrow$$

```
for i in range(5000):
    optimiser.zero_grad()

    lambda1, lambda2, lambda3 = 1,1e-4,1e-1

    # compute initial data loss
    u = pinn(initial_data_nodes_t)
    loss1 = torch.mean((torch.squeeze(u) - initial_data_values_t)**2)

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss2 = torch.mean((u - torch.zeros(u.shape))**2)

    # compute physics loss
    u = pinn(collocation_data_nodes_t)
    du = torch.autograd.grad(u, collocation_data_nodes_t,\
                             torch.ones([collocation_data_nodes_t.shape[0], 1]),\
                             retain_graph=True, create_graph=True)[0]
    du2 = torch.autograd.grad(du, collocation_data_nodes_t,\
                              torch.ones(collocation_data_nodes_t.shape),\
                              create_graph=True)[0]
    u_x  = du[:,[0]]
    u_t  = du[:,[1]]
    u_xx = du2[:,[0]]
    loss3 = torch.mean((u_t + u*u_x - .01 / np.pi *u_xx)**2)

    # backpropagate joint loss, take optimiser step
    loss = lambda1*loss1 + lambda2*loss2 + lambda1*loss3
    loss.backward()
    optimiser.step()
```

PDE

Training cycle:

$$| \mathcal{N}\mathcal{N}(x_i) - u_0(x_i) | \longrightarrow$$

$$| \mathcal{N}\mathcal{N}(-1,t_i) - 0 + \mathcal{N}\mathcal{N}(1,t_i) - 0 |^2 \longrightarrow$$

```
for i in range(5000):
    optimiser.zero_grad()

    lambda1, lambda2, lambda3 = 1,1e-4,1e-1

    # compute initial data loss
    u = pinn(initial_data_nodes_t)
    loss1 = torch.mean((torch.squeeze(u) - initial_data_values_t)**2)

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss2 = torch.mean((u - torch.zeros(u.shape))**2)

    # compute physics loss
    u = pinn(collocation_data_nodes_t)
    du = torch.autograd.grad(u, collocation_data_nodes_t,\
                             torch.ones([collocation_data_nodes_t.shape[0], 1]),\
                             retain_graph=True, create_graph=True)[0]
    du2 = torch.autograd.grad(du, collocation_data_nodes_t,\
                              torch.ones(collocation_data_nodes_t.shape),\
                              create_graph=True)[0]
    u_x  = du[:,[0]]
    u_t  = du[:,[1]]
    u_xx = du2[:,[0]]
    loss3 = torch.mean((u_t + u*u_x - .01 / np.pi *u_xx)**2)

    # backpropagate joint loss, take optimiser step
    loss  = lambda1*loss1 + lambda2*loss2 + lambda1*loss3
    loss.backward()
    optimiser.step()
```

PDE

Training cycle:

$$|\mathcal{N}\mathcal{N}(x_i) - u_0(x_i)|$$

$$|\mathcal{N}\mathcal{N}(-1,t_i) - 0 + \mathcal{N}\mathcal{N}(1,t_i) - 0|^2$$

$$\partial_t u, \partial_x u, \partial_{xx} u$$

```
for i in range(5000):
    optimiser.zero_grad()

    lambda1, lambda2, lambda3 = 1,1e-4,1e-1

    # compute initial data loss
    u = pinn(initial_data_nodes_t)
    loss1 = torch.mean((torch.squeeze(u) - initial_data_values_t)**2)

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss2 = torch.mean((u - torch.zeros(u.shape))**2)

    # compute physics loss
    u = pinn(collocation_data_nodes_t)
    du = torch.autograd.grad(u, collocation_data_nodes_t,\
                             torch.ones([collocation_data_nodes_t.shape[0], 1]),\
                             retain_graph=True, create_graph=True)[0]
    du2 = torch.autograd.grad(du, collocation_data_nodes_t,\
                              torch.ones(collocation_data_nodes_t.shape),\
                              create_graph=True)[0]
    u_x  = du[:,[0]]
    u_t  = du[:,[1]]
    u_xx = du2[:,[0]]
    loss3 = torch.mean((u_t + u*u_x - .01 / np.pi *u_xx)**2)

    # backpropagate joint loss, take optimiser step
    loss = lambda1*loss1 + lambda2*loss2 + lambda1*loss3
    loss.backward()
    optimiser.step()
```

PDE

Training cycle:

$$|\mathcal{N}\mathcal{N}(x_i) - u_0(x_i)|$$

$$|\mathcal{N}\mathcal{N}(-1,t_i) - 0 + \mathcal{N}\mathcal{N}(1,t_i) - 0|^2$$

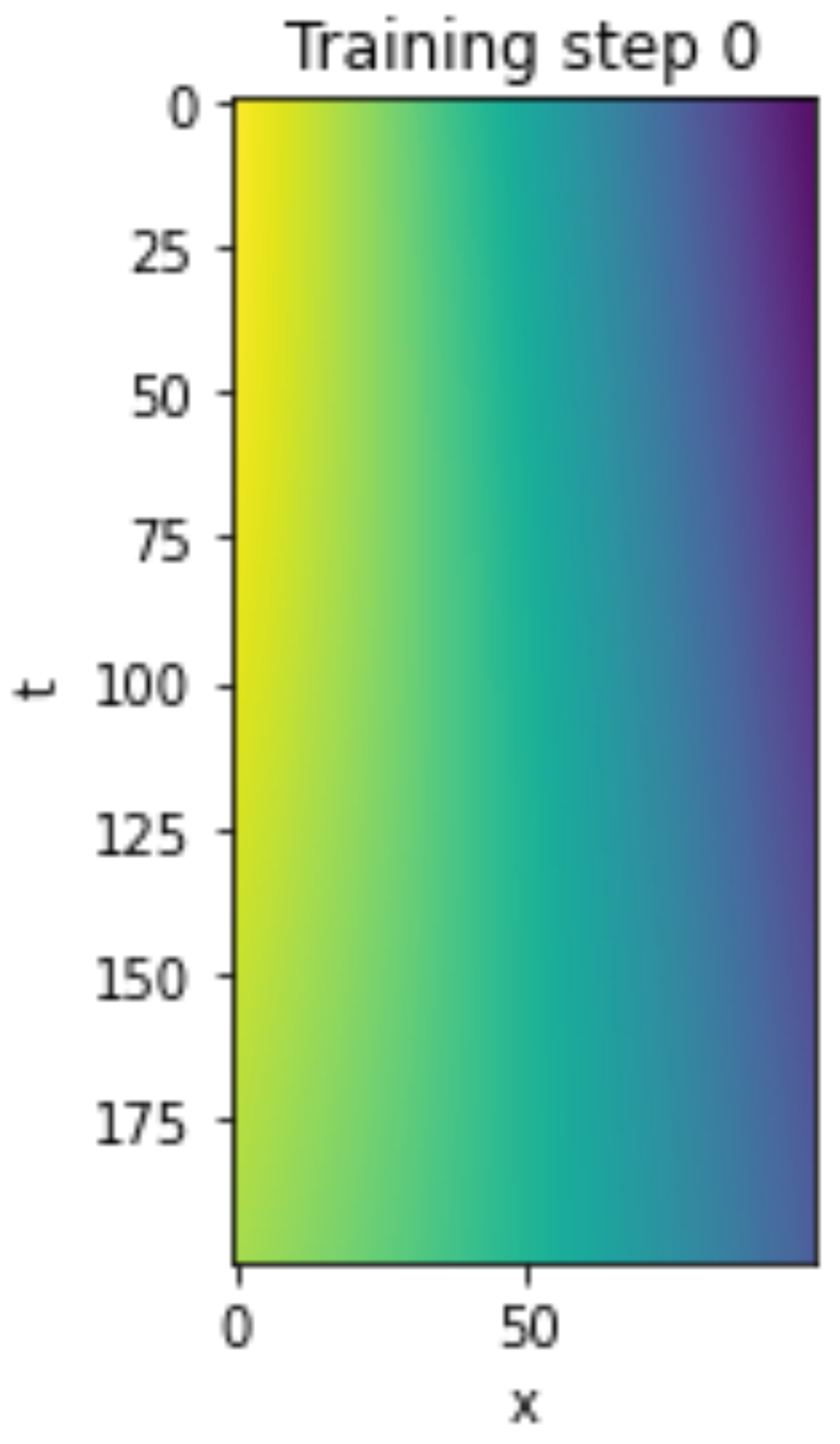
$$\partial_t u, \partial_x u, \partial_{xx} u$$

$$|f(u, x_i, t_i)|^2$$

```python
for i in range(5000):
    optimiser.zero_grad()

    lambda1, lambda2, lambda3 = 1,1e-4,1e-1

    # compute initial data loss
    u = pinn(initial_data_nodes_t)
    loss1 = torch.mean((torch.squeeze(u) - initial_data_values_t)**2)

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss2 = torch.mean((u - torch.zeros(u.shape))**2)

    # compute physics loss
    u = pinn(collocation_data_nodes_t)
    du = torch.autograd.grad(u, collocation_data_nodes_t,\
                             torch.ones([collocation_data_nodes_t.shape[0], 1]),\
                             retain_graph=True, create_graph=True)[0]
    du2 = torch.autograd.grad(du, collocation_data_nodes_t,\
                              torch.ones(collocation_data_nodes_t.shape),\
                              create_graph=True)[0]
    u_x  = du[:,[0]]
    u_t  = du[:,[1]]
    u_xx = du2[:,[0]]
    loss3 = torch.mean((u_t + u*u_x - .01 / np.pi *u_xx)**2)

    # backpropagate joint loss, take optimiser step
    loss = lambda1*loss1 + lambda2*loss2 + lambda1*loss3
    loss.backward()
    optimiser.step()
```

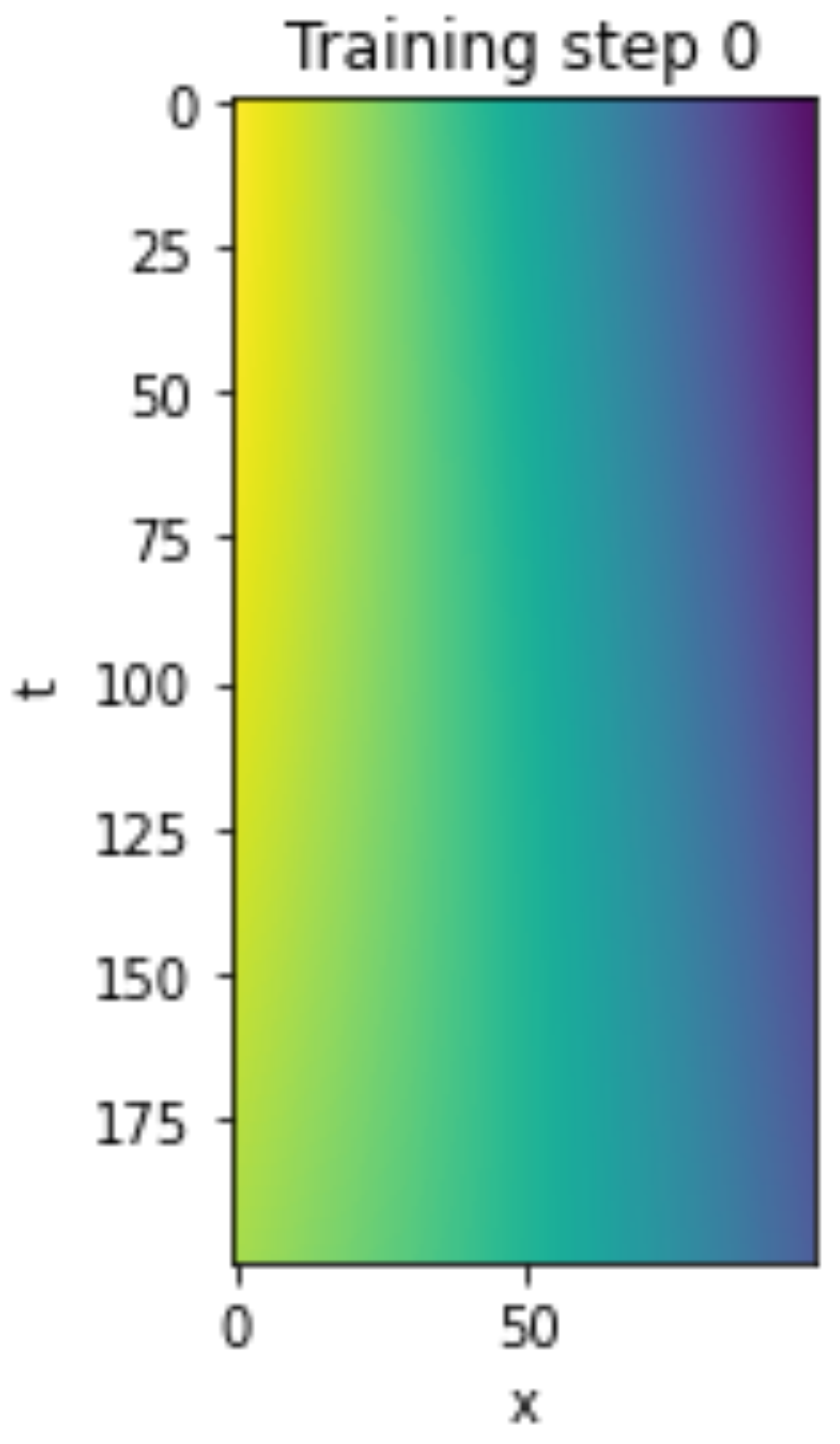
PDE

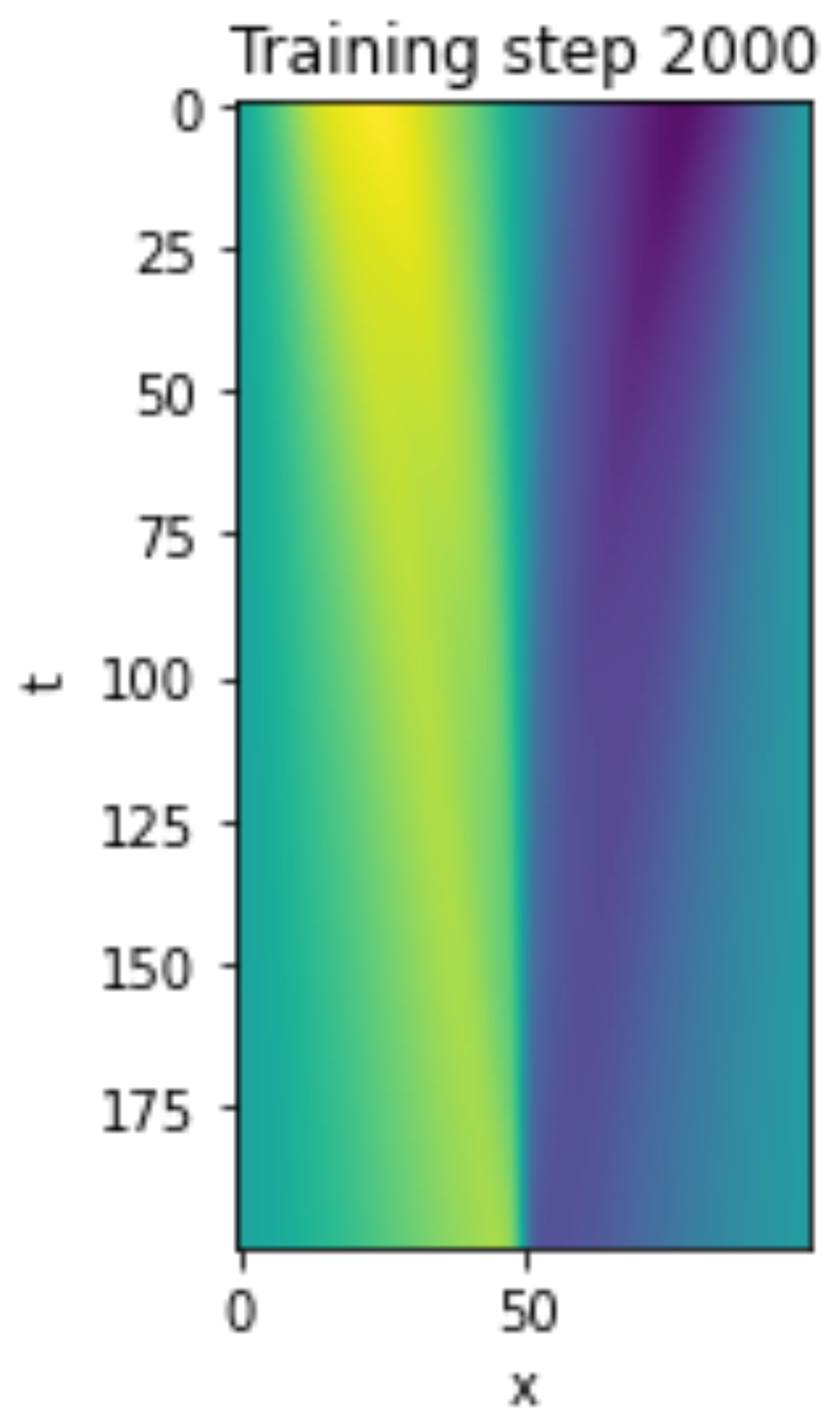
Training cycle:



PDE

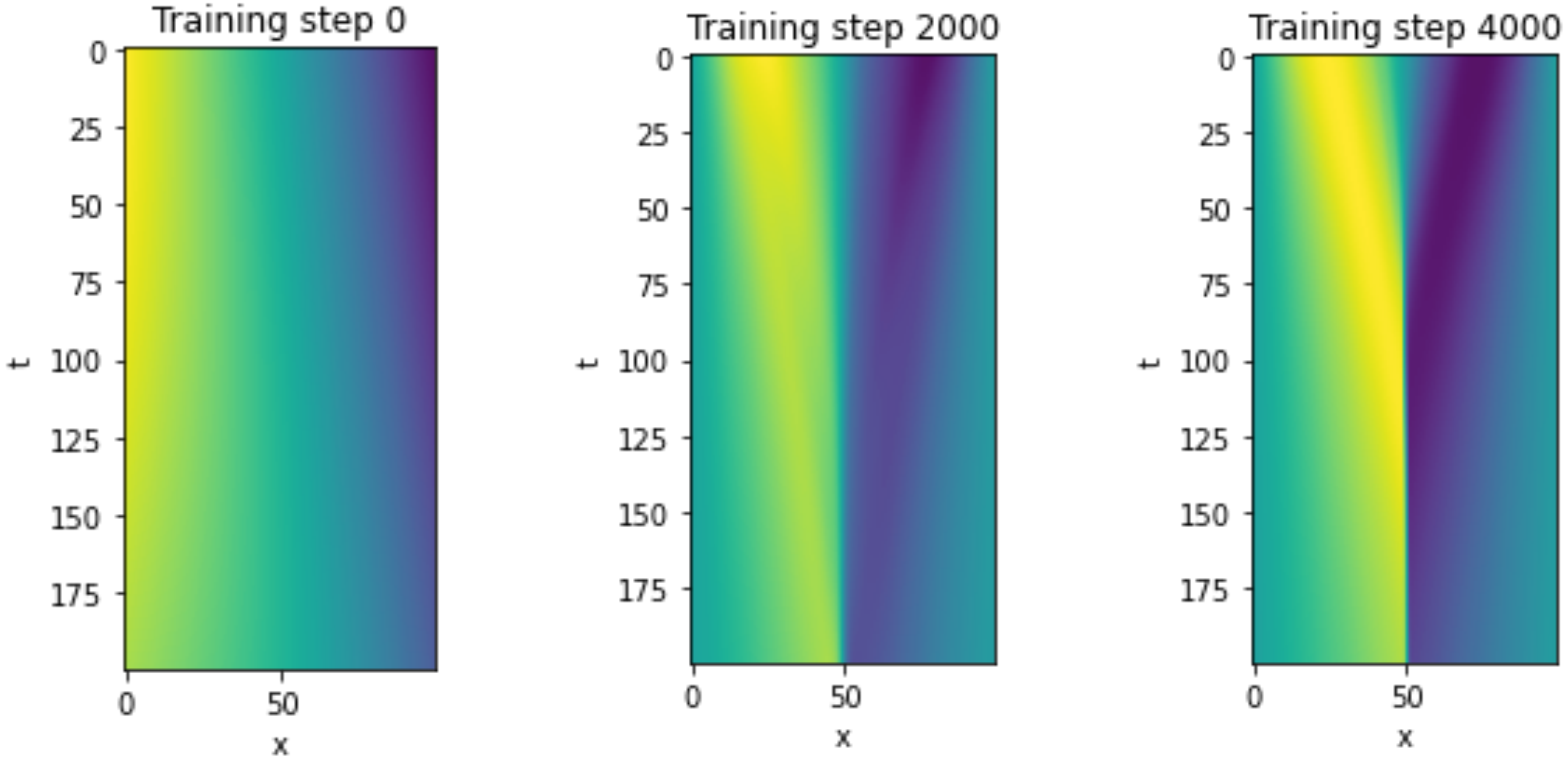
Training cycle:





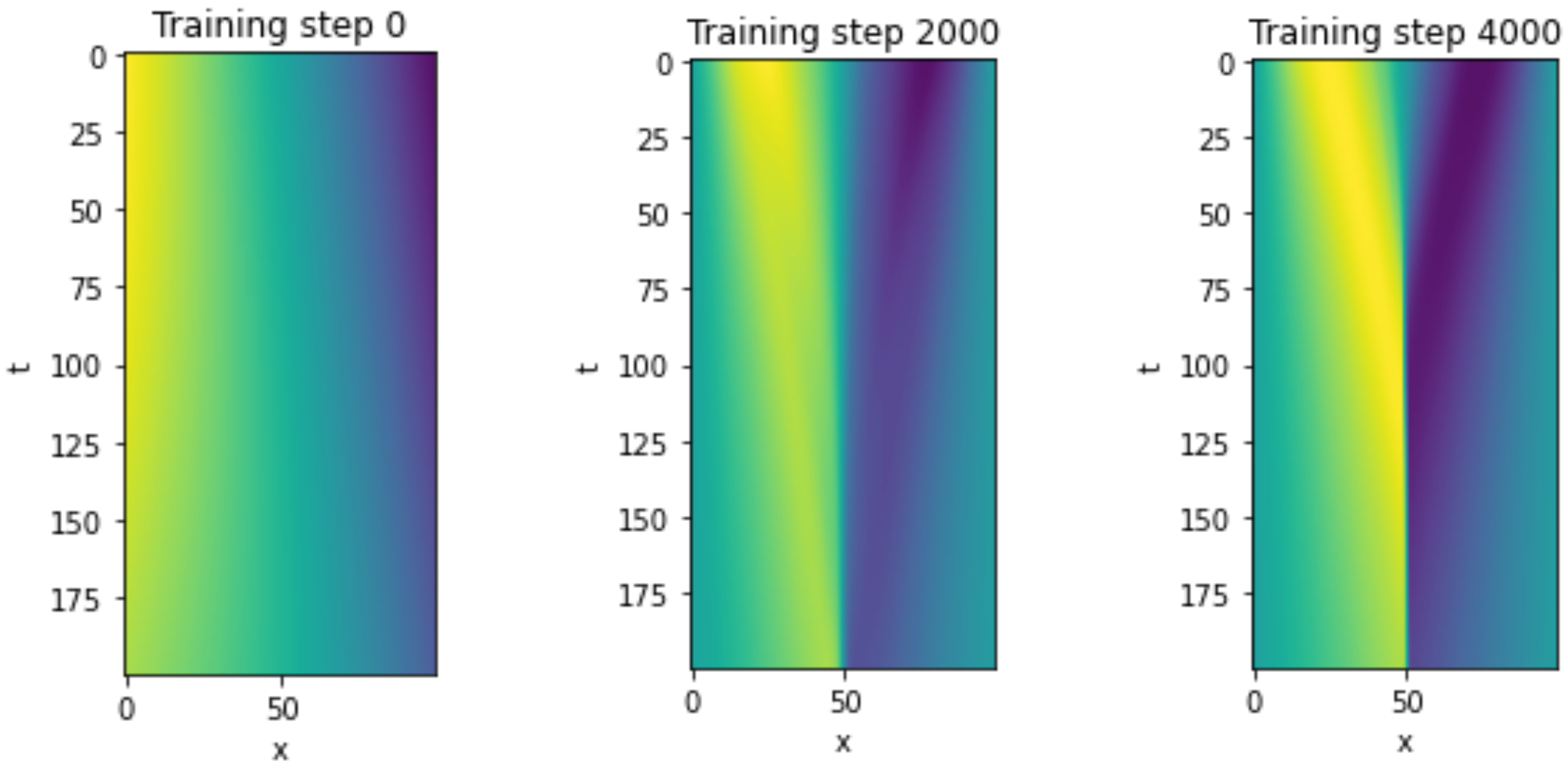
PDE

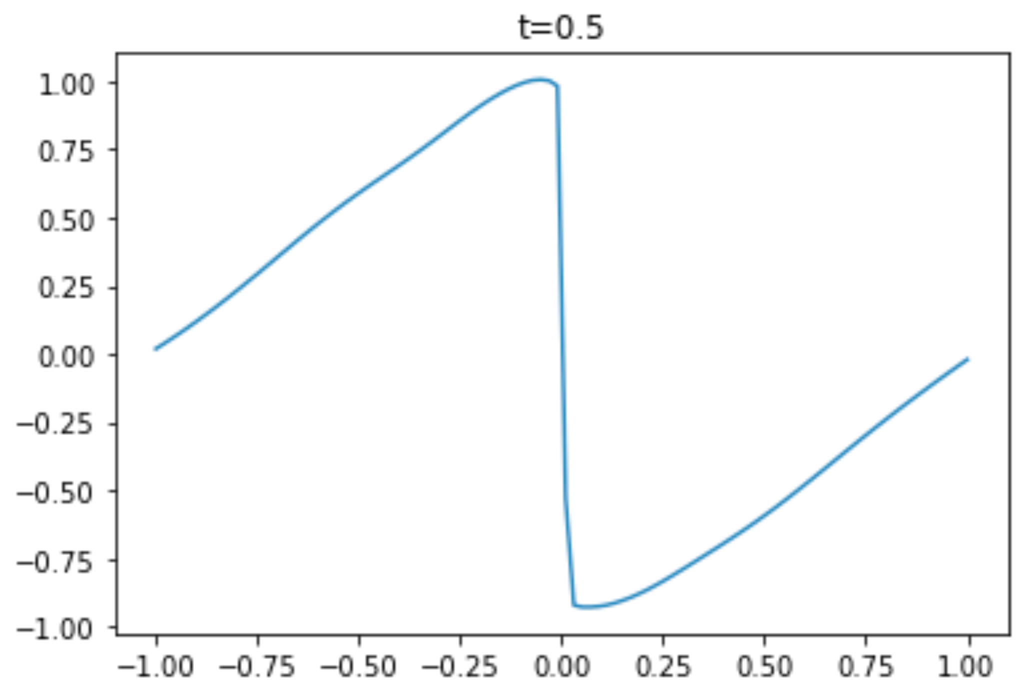
Training cycle:



PDE

Training cycle:





ODE

- The displacement of the spring follows the following differential equation:

$$m\frac{d^2}{dt^2}u + \mu\frac{d}{dt}u + ku = 0$$

m is the mass of the oscillator, μ the coefficient of friction and k the spring constant.

Considering the case where the oscillations are damped by friction and initial conditions of the system are $u(t=0) = 1, \quad \frac{d}{dt}u(t=0) = 0$.

There is an analytical solution for this problem.

https://beltoforion.de/en/harmonic_oscillator/

ODE

- The displacement of the spring follows the following differential equation:

$$m\frac{d^2}{dt^2}u + \mu\frac{d}{dt}u + ku = 0$$

m is the mass of the oscillator, μ the coefficient of friction and k the spring constant.

Considering the case where the oscillations are damped by friction and initial conditions of the system are $u(t=0)=1, \quad \frac{d}{dt}u(t=0)=0.$

There is an analytical solution for this problem.

https://beltoforion.de/en/harmonic_oscillator/

Use PINNs to solve this ODE

ODE

Define the sampling nodes

```
boundary_nodes = np.zeros(1)
collocation_nodes = np.linspace(0,1,30)
test_nodes = np.linspace(0,1,300)

boundary_nodes_t = torch.tensor(boundary_nodes,dtype=torch.float).view(-1,1)
collocation_nodes_t = torch.tensor(collocation_nodes,dtype=torch.float).view(-1,1)
test_nodes_t = torch.tensor(test_nodes,dtype=torch.float).view(-1,1)

boundary_nodes_t.requires_grad = True
collocation_nodes_t.requires_grad = True
test_nodes_t.requires_grad = True
```

- Source: https://github.com/benmoseley/DLSC-2023

ODE

Neural network architecture the same as for PDE (except input is dimension 1).

Training cycle:

$\mathscr{L}_b \longrightarrow$

```
for i in range(15001):
    optimiser.zero_grad()

    lambda1, lambda2 = 1e-1, 1e-4

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss1 = (torch.squeeze(u) - 1)**2
    dudt = torch.autograd.grad(u, boundary_nodes_t,\
                               torch.ones_like(u),retain_graph=True, create_graph=True)[0]
    loss2 = (torch.squeeze(dudt) - 0)**2

    # compute physics loss
    u = pinn(collocation_nodes_t)
    dudt = torch.autograd.grad(u, collocation_nodes_t,\
                               torch.ones_like(u), create_graph=True)[0]
    d2udt2 = torch.autograd.grad(dudt, collocation_nodes_t,\
                                 torch.ones_like(dudt), create_graph=True)[0]
    loss3 = torch.mean((d2udt2 + mu*dudt + k*u)**2)

    # backpropagate joint loss, take optimiser step
    loss = loss1 + lambda1*loss2 + lambda2*loss3
    loss.backward()
    optimiser.step()
```

ODE

Neural network architecture the same as for PDE (except input is dimension 1).

Training cycle:

$\mathscr{L}_b$

$$\frac{d}{dt}u, \frac{d^2}{dt^2}u$$

```
for i in range(15001):
    optimiser.zero_grad()

    lambda1, lambda2 = 1e-1, 1e-4

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss1 = (torch.squeeze(u) - 1)**2
    dudt = torch.autograd.grad(u, boundary_nodes_t,\
                               torch.ones_like(u),retain_graph=True, create_graph=True)[0]
    loss2 = (torch.squeeze(dudt) - 0)**2

    # compute physics loss
    u = pinn(collocation_nodes_t)
    dudt = torch.autograd.grad(u, collocation_nodes_t,\
                               torch.ones_like(u), create_graph=True)[0]
    d2udt2 = torch.autograd.grad(dudt, collocation_nodes_t,\
                                 torch.ones_like(dudt), create_graph=True)[0]
    loss3 = torch.mean((d2udt2 + mu*dudt + k*u)**2)

    # backpropagate joint loss, take optimiser step
    loss = loss1 + lambda1*loss2 + lambda2*loss3
    loss.backward()
    optimiser.step()
```

ODE

Neural network architecture the same as for PDE (except input is dimension 1).

Training cycle:

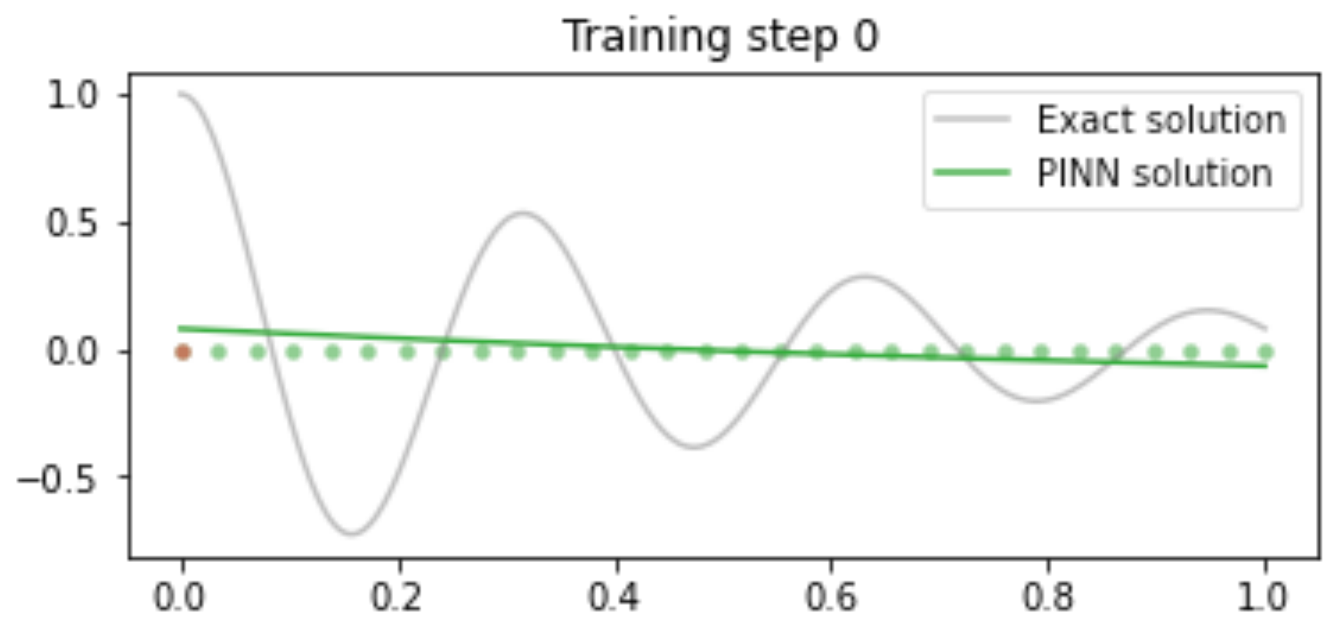
$\mathcal{L}_b$

$\frac{d}{dt}u, \frac{d^2}{dt^2}u$

$\mathcal{L}_f$

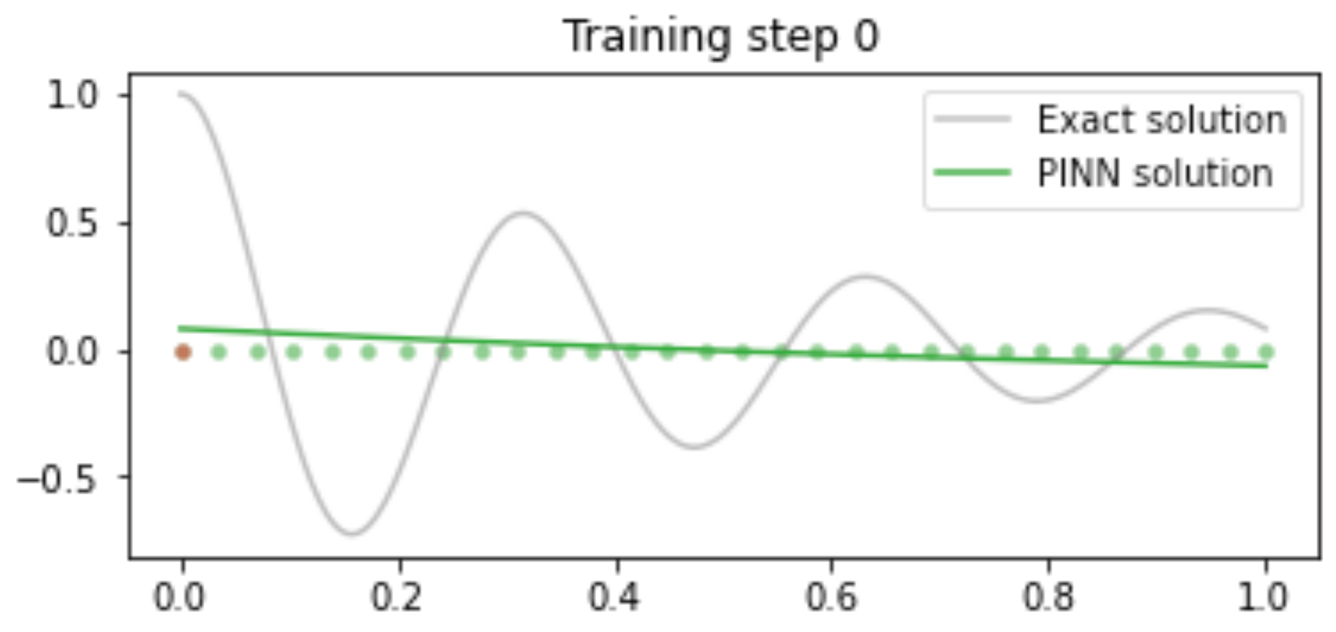
```
for i in range(15001):
    optimiser.zero_grad()

    lambda1, lambda2 = 1e-1, 1e-4

    # compute boundary loss
    u = pinn(boundary_nodes_t)
    loss1 = (torch.squeeze(u) - 1)**2
    dudt = torch.autograd.grad(u, boundary_nodes_t,\
                               torch.ones_like(u),retain_graph=True, create_graph=True)[0]
    loss2 = (torch.squeeze(dudt) - 0)**2

    # compute physics loss
    u = pinn(collocation_nodes_t)
    dudt = torch.autograd.grad(u, collocation_nodes_t,\
                               torch.ones_like(u), create_graph=True)[0]
    d2udt2 = torch.autograd.grad(dudt, collocation_nodes_t,\
                                 torch.ones_like(dudt), create_graph=True)[0]
    loss3 = torch.mean((d2udt2 + mu*dudt + k*u)**2)

    # backpropagate joint loss, take optimiser step
    loss = loss1 + lambda1*loss2 + lambda2*loss3
    loss.backward()
    optimiser.step()
```

ODE

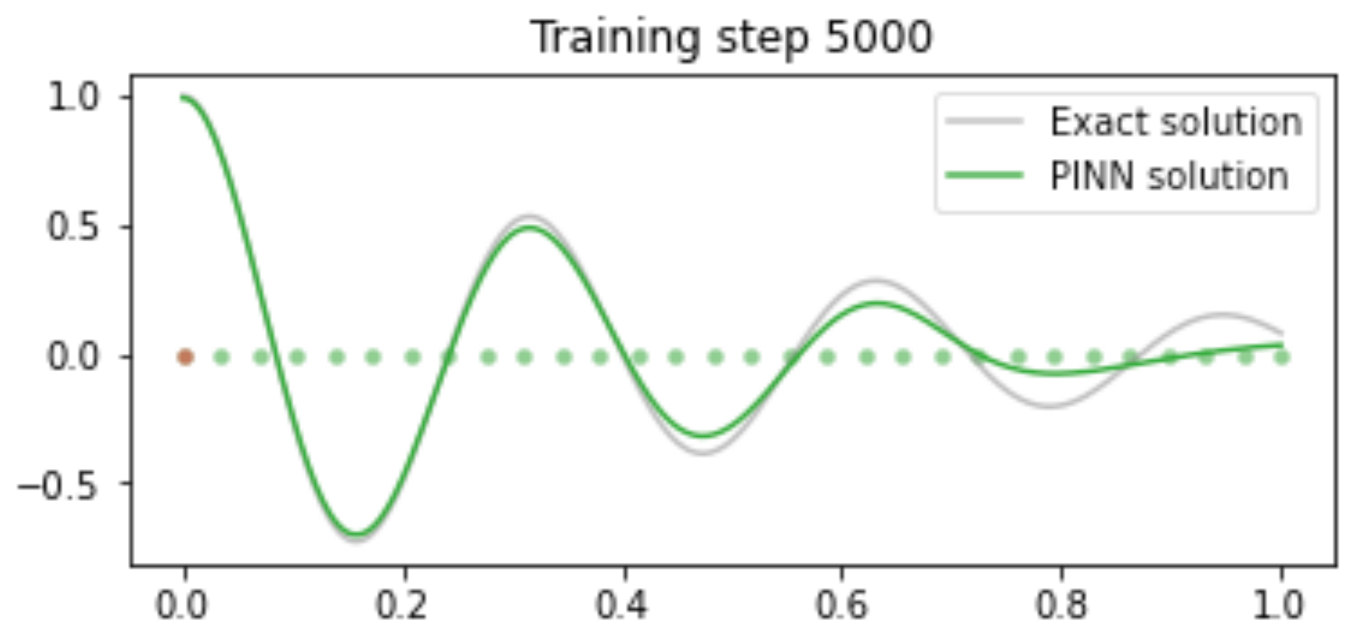


Training step 0
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0

ODE

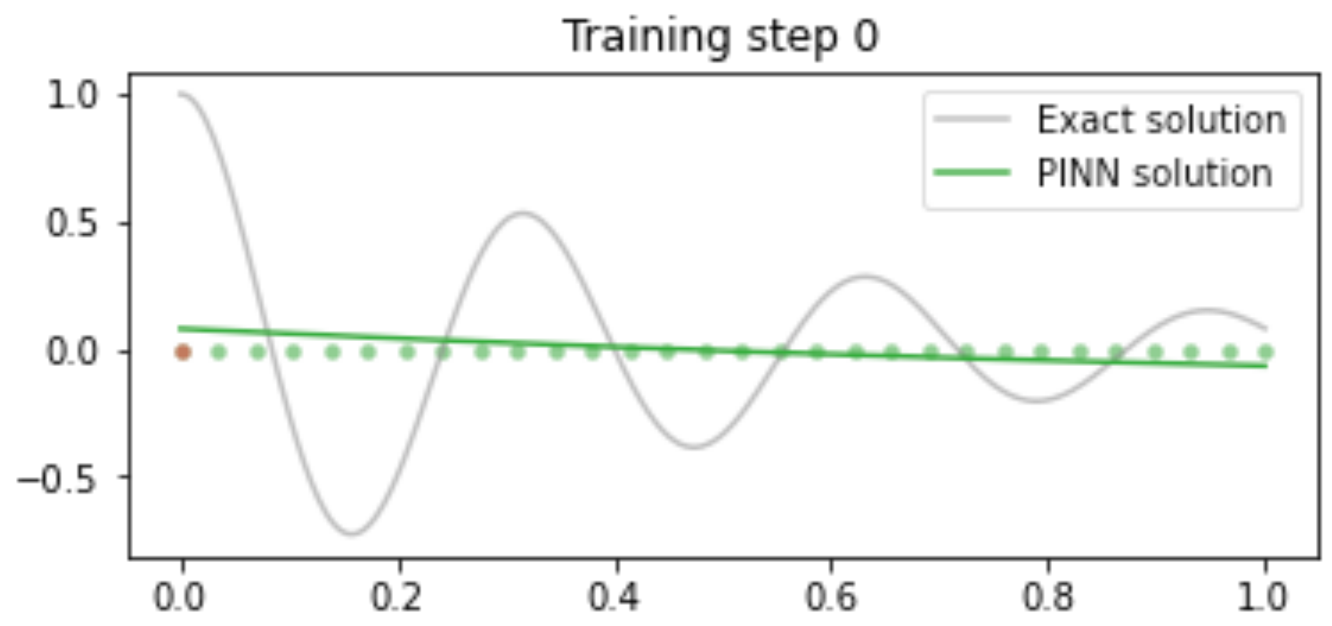


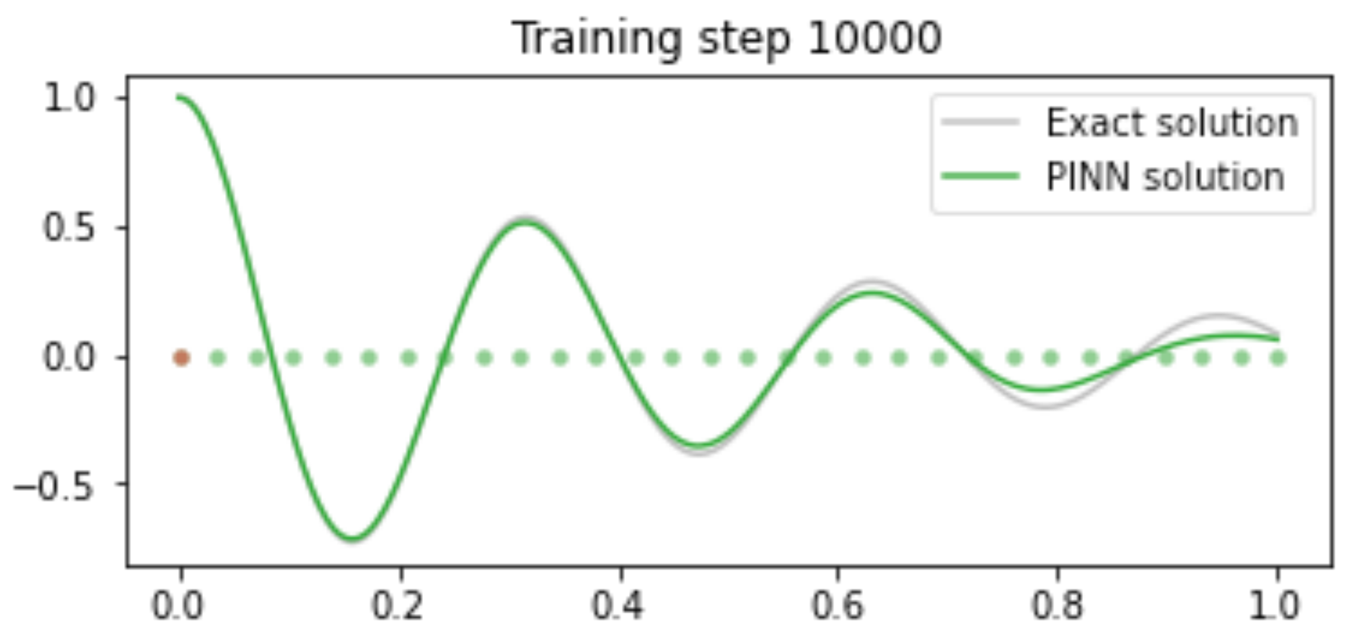
Training step 0
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0

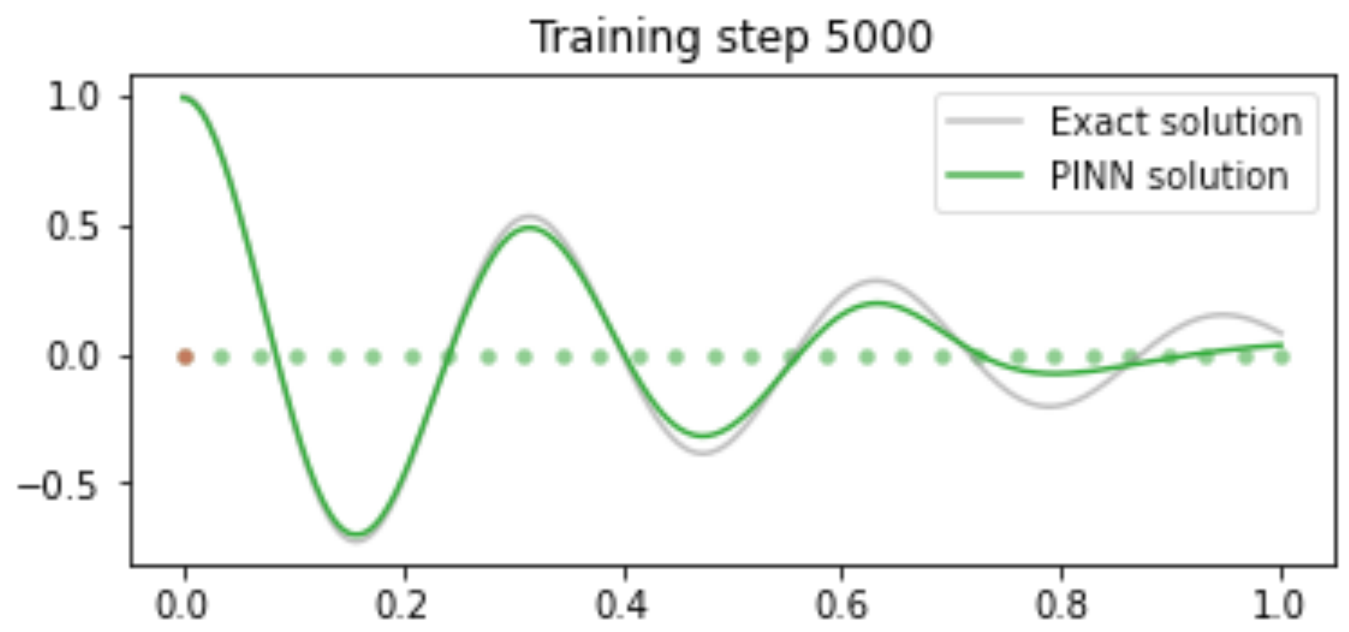


Training step 5000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0

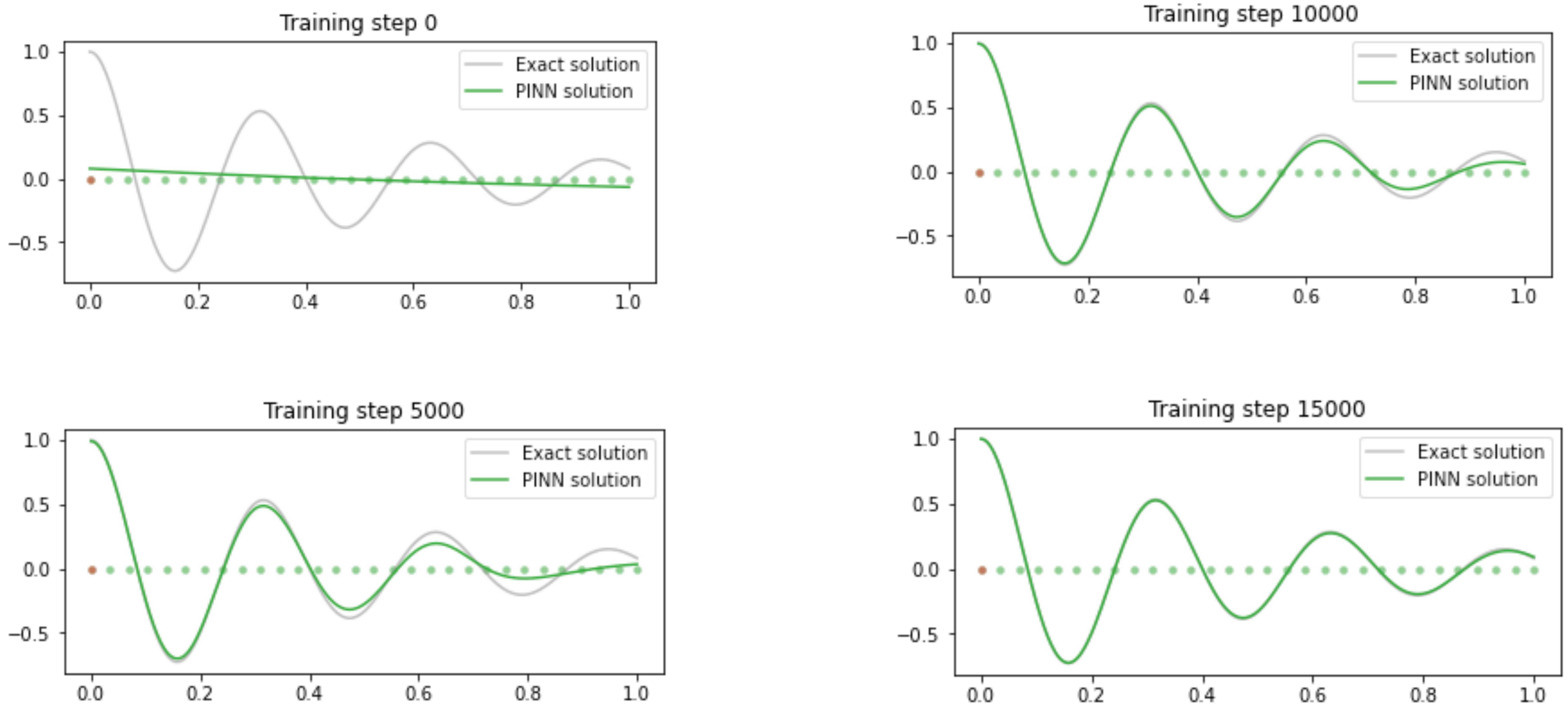
ODE


Training step 0
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 10000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 5000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0

ODE


Training step 0
Exact solution
PINN solution
Training step 10000
Exact solution
PINN solution
Training step 5000
Exact solution
PINN solution
Training step 15000
Exact solution
PINN solution