PINN Training

Tutorial 3

Physics-informed neural networks for the 1D heat equation: residuals, loss and training in PyTorch.

Bogdan Raonic · Oct 5, 2026 · Seminar for Applied Mathematics (D-MATH) & ETH AI Center

Outline

01 From PDE to loss

02 Implementation in PyTorch

03 Results and common mistakes

04 Practice

01

From PDE to loss

PINNs: from a network to a loss

Goal: find weights θ so that the network $u_\theta(t, x)$ satisfies the PDE, the boundary condition and the initial condition. No solution values are needed inside the domain.

01 · Network

$(t, x) \mapsto u_\theta(t, x)$

Input: a point in space and time. Output: the network's guess for u at that point.

02 · Autograd

$\partial_t u_\theta,\ \partial_{xx} u_\theta$

u_θ is a smooth function of its input, so PyTorch differentiates it in t and x exactly. No grid, no finite differences.

03 · Residuals

r = left side − right side

Plug u_θ and its derivatives into each equation. The residual measures how much the equation is violated; zero means satisfied.

04 · Loss

$L(\theta)$ = mean of r^2

Average the squared residuals over the training points and minimize over θ. L = 0 means u_θ solves the problem.

EXAMPLE $u_t = u_{xx}$ gives $r = \partial_t u_\theta - \partial_{xx} u_\theta$, and $u(0, x) = -\sin(\pi x)$ gives $r = u_\theta(0, x) + \sin(\pi x)$

The 1D heat equation

PDE $u_t = u_{xx}$

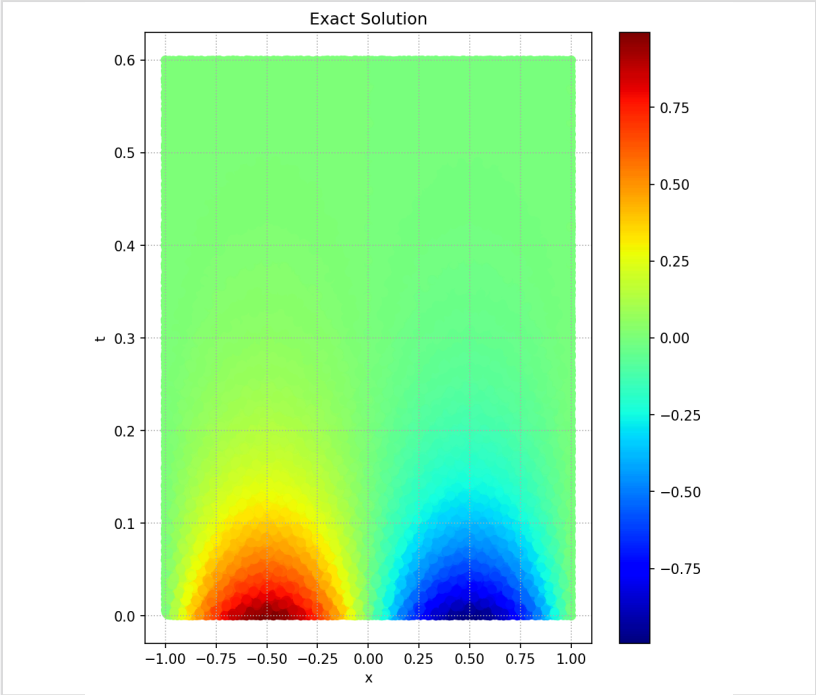
$t \in [0, 0.6], x \in [-1, 1]$

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

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

EXACT $u(t, x) = -e^{-\pi^2 t} \sin(\pi x)$

Used only to measure the error



Residuals and loss

Interior

$$r_{\text{int}} = \partial_t u_\theta - \partial_{xx} u_\theta$$

on S_{int}: points inside $[0, T] \times [-1, 1]$

Spatial boundary

$$r_{\text{sb}} = u_\theta(t, \pm 1) - u_b(t, \pm 1)$$

on S_{sb}: points on $x = -1$ and $x = 1$

Temporal boundary

$$r_{\text{tb}} = u_\theta(0, x) - u_0(x)$$

on S_{tb}: points on $t = 0$

Loss

$$L(\theta) = L_{\text{int}} + \lambda_u (L_{\text{sb}} + L_{\text{tb}})$$

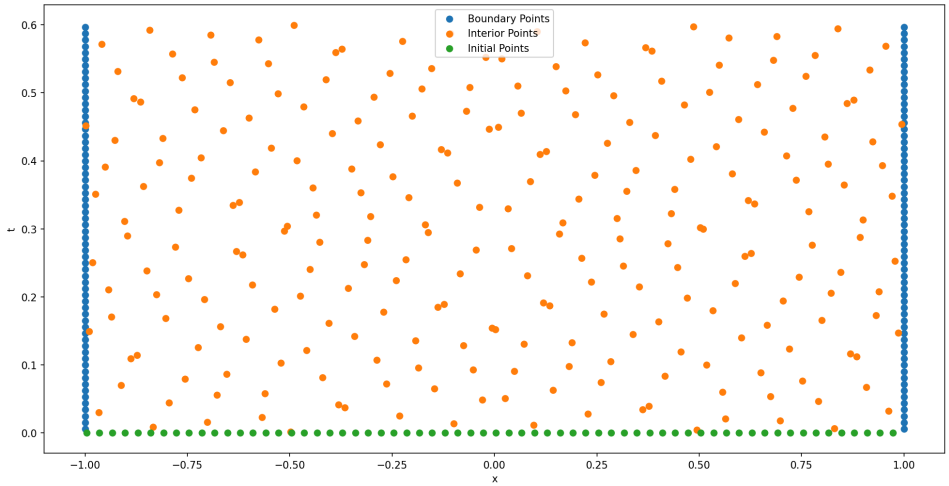
each L is the mean squared residual on its set: a quadrature rule for the integrals in the lecture

02

Implementation

Training points: a Sobol sequence

Sobol points: a sequence of points that covers the domain more evenly than random sampling.



256

interior points, for the PDE residual

2 × 64

boundary points at x = ±1, target u = 0

64

initial points at t = 0, target −sin(πx)

The network: a tanh MLP

```
self.approximate_solution = NeuralNet(
    input_dimension=2,    # (t, x)
    output_dimension=1,   # u
    n_hidden_layers=4,
    neurons=20,
    regularization_param=0.,
    regularization_exp=2.,
    retrain_seed=42)

def init_weights(m):
    g = nn.init.calculate_gain('tanh')
    nn.init.xavier_uniform_(m.weight, gain=g)
    m.bias.data.fill_(0)
```

Two inputs, one output

The same kind of MLP as in Tutorial 1, now with (t, x) as input.

tanh, not ReLU

The residual needs u_{xx}. The second derivative of ReLU is zero; tanh is smooth.

Xavier initialization

Weights scaled for tanh, biases set to zero. A fixed seed makes runs repeatable.

PDE residual: two autograd calls

```
def compute_pde_residual(self, input_int):
    input_int.requires_grad = True
    u = self.approximate_solution(input_int)

    grad_u = torch.autograd.grad(
        u.sum(), input_int, create_graph=True)[0]
    grad_u_t = grad_u[:, 0]
    grad_u_x = grad_u[:, 1]
    grad_u_xx = torch.autograd.grad(
        grad_u_x.sum(), input_int,
        create_graph=True)[0][:, 1]

    return (grad_u_t - grad_u_xx).reshape(-1, )
```

requires_grad on the input

Derivatives in t and x, not in the weights. Set it before the forward pass.

u.sum()

Each u_i depends only on its own point, so one gradient gives every $\partial u_i / \partial x_i$.

create_graph=True

Needed to differentiate u_x again, and to backpropagate the loss through the residual.

Loss: weighted, on a log scale

```
r_int = self.compute_pde_residual(inp_train_int)
r_sb = u_train_sb - u_pred_sb
r_tb = u_train_tb - u_pred_tb

loss_sb = torch.mean(abs(r_sb) ** 2)
loss_tb = torch.mean(abs(r_tb) ** 2)
loss_int = torch.mean(abs(r_int) ** 2)

loss = torch.log10(
    self.lambda_u * (loss_sb + loss_tb) + loss_int)
```

Mean = quadrature

Each mean replaces an integral. Sobol points make it a quasi Monte Carlo rule.

$\lambda_u = 10$

Weights the boundary and initial data against the PDE residual.

log10

Same minimizer, better scaled. The printed loss goes from about 1.5 to −5.3.

Optimizer: L-BFGS with a closure

```
optimizer_LBFGS = optim.LBFGS(
    pinn.approximate_solution.parameters(),
    lr=0.5,
    max_iter=50000,
    history_size=150,
    line_search_fn="strong_wolfe")

def closure():
    optimizer.zero_grad()
    loss = self.compute_loss(...)
    loss.backward()
    return loss

optimizer.step(closure=closure)
```

Full batch

All 384 points in one batch. With so few points, a quasi-Newton method pays off.

closure

The line search evaluates the loss several times per step, so step() needs a function.

One epoch

A single step() runs up to 50 000 iterations. Adam with lr = 1e-3 is in the notebook to compare.

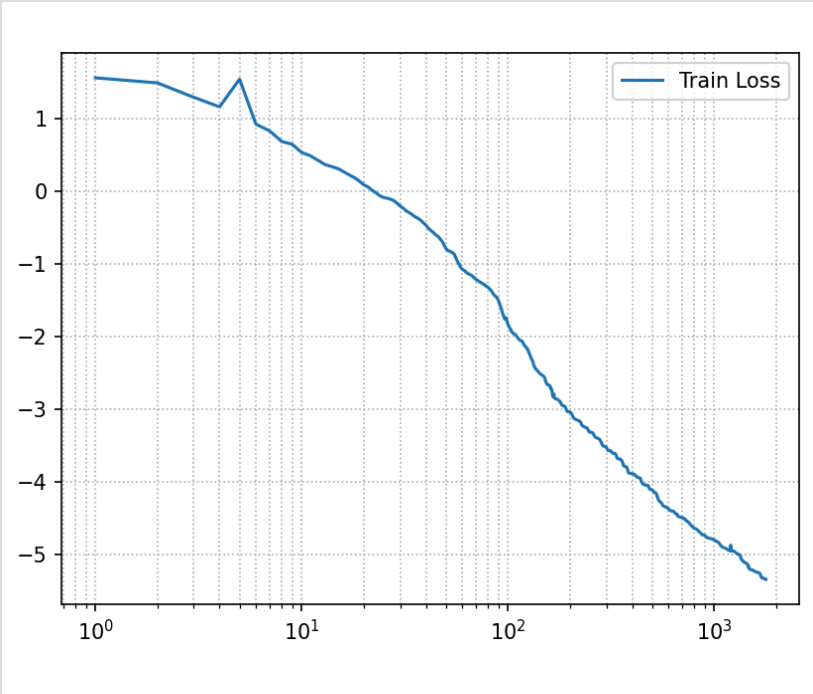
03

Results

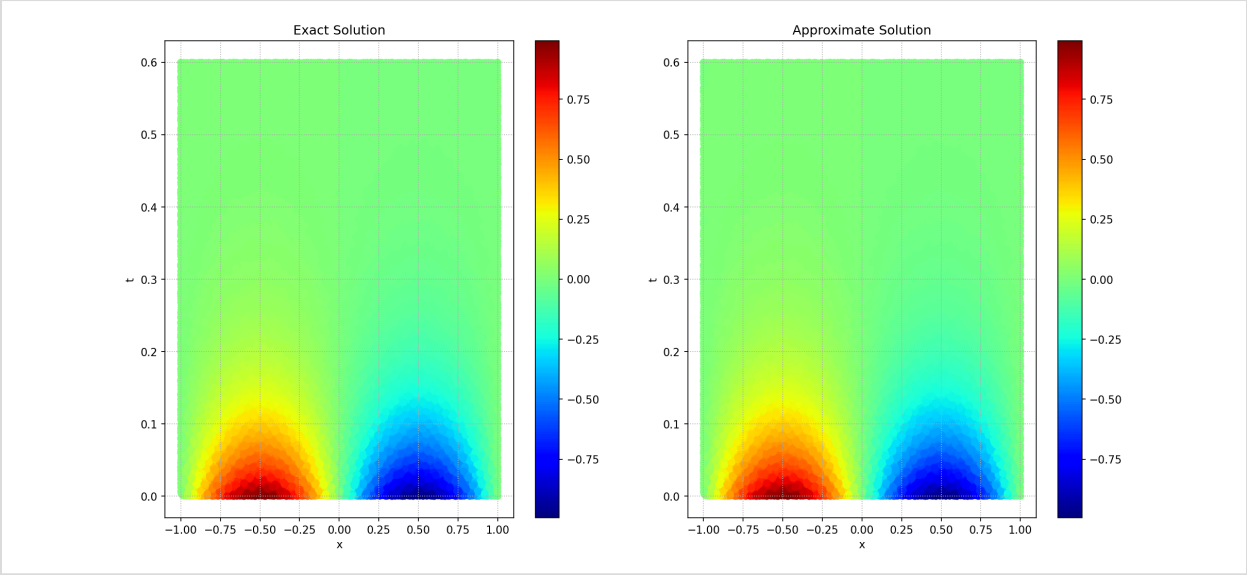
Training: L-BFGS loss

log10 of the total loss, from 1.56 at the first iteration to −5.35 after about 1 800 iterations.

The curve is already logarithmic: the loss falls by almost seven orders of magnitude.



Result: exact and PINN solution



0.06%

relative L2 error, measured on 100 000 Sobol points

Network 4 × 20, 384 training points, no solution data inside the domain.

Common mistakes: four PINN bugs

WHAT YOU SEE	CAUSE	FIX
Error: tensor does not require grad	requires_grad not set on the inputs	Set it before the forward pass
Error on the second grad call	create_graph=False in the first call	Pass create_graph=True every time
u_xx is zero everywhere	ReLU activation	Use a smooth activation, e.g. tanh
PDE is satisfied, but the solution is wrong	λ_u too small, data terms ignored	Increase λ_u, print each loss term

04

Practice

Practice: the heat equation on Colab

- Check link on the Moodle.
- Run the notebook as is: L-BFGS reaches about 0.06 % error.
- Swap L-BFGS for Adam and compare the loss curves.
- Change λ_u and the number of collocation points.