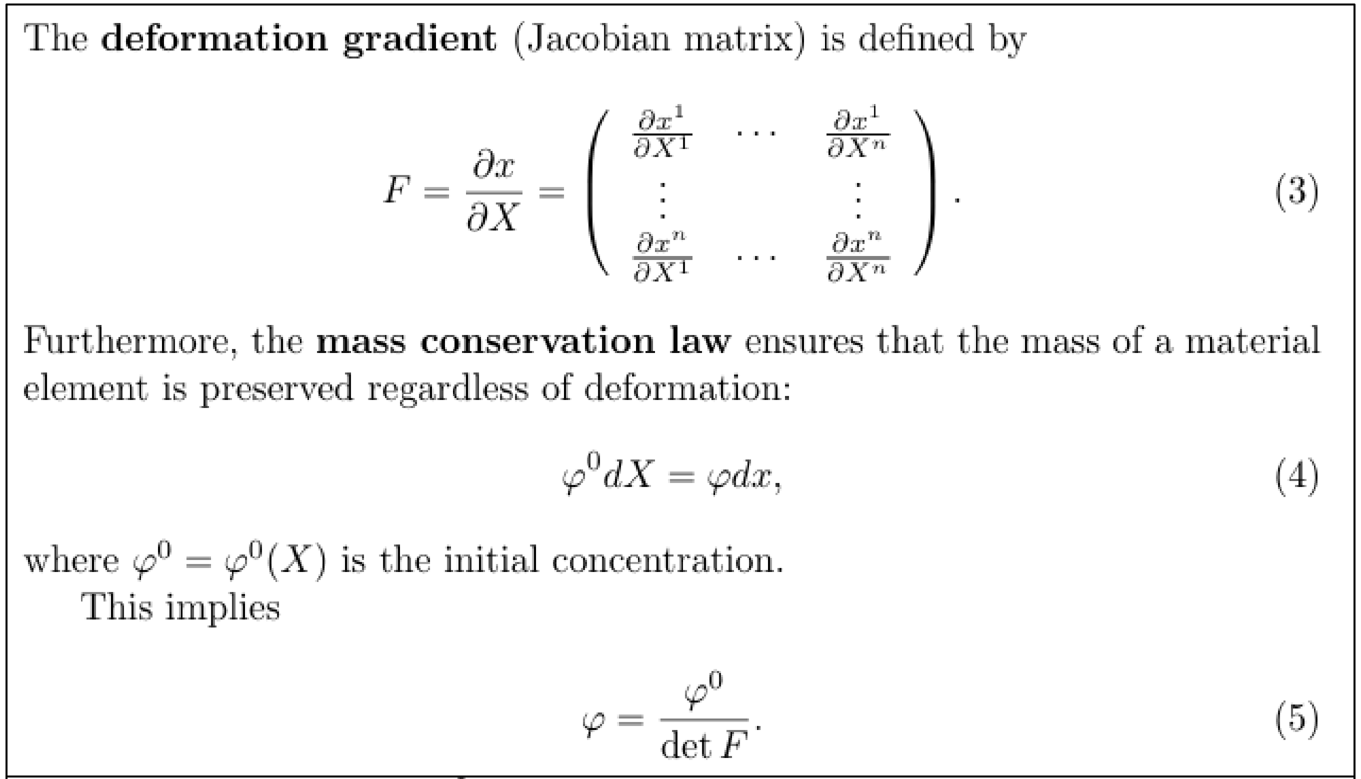
Neural Constitutive Laws for Diffusion Equation

PDE

- 1D heat diffusion: $\rho_t = \mathrm{c}\Delta\rho,\ (x,t) \in [0,1] \times [0,1],$ c=1
- Initial condition: $\rho(x,0) = \sin(\pi x)$
- Boundary condition: $\rho(0,t) = \rho(1,t) = 0$
- Exact solution: $\rho_{exact}(x,t) = \sin(\pi x)\,\mathrm{e}^{-\pi^2 t}$
- Number of grid points: $\mathrm{NX} = 101, \mathrm{NT} = 101\ (includes\ endpoints)$
- Grid size: $\Delta\mathrm{x} = \frac{1}{100}, \Delta t = \frac{1}{100}$

Notice that PDE consists of two parts:

- **Conservation law**: $\rho_t + \nabla \cdot (\rho u) = 0 \Leftrightarrow \rho J = \rho_0$
- **Constitutive law**: $\rho u = -\nabla \rho$



The **deformation gradient** (Jacobian matrix) is defined by

$$F = \frac{\partial x}{\partial X} = \begin{pmatrix} \frac{\partial x^1}{\partial X^1} & \cdots & \frac{\partial x^1}{\partial X^n} \\ \vdots & & \vdots \\ \frac{\partial x^n}{\partial X^1} & \cdots & \frac{\partial x^n}{\partial X^n} \end{pmatrix}. \tag{3}$$

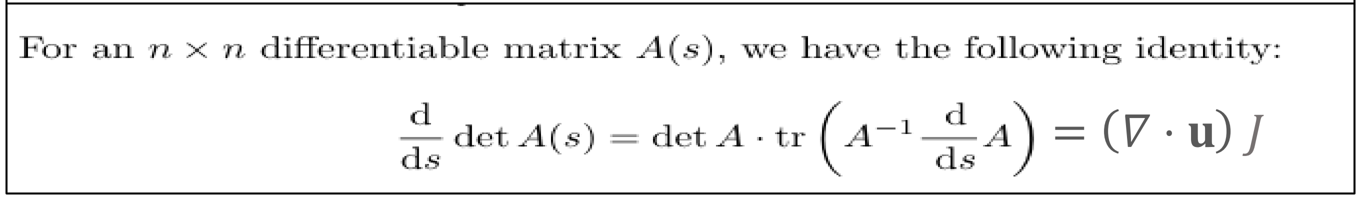
Furthermore, the **mass conservation law** ensures that the mass of a material element is preserved regardless of deformation:

$$\varphi^0 dX = \varphi dx, \tag{4}$$

where $\varphi^0 = \varphi^0(X)$ is the initial concentration.

This implies

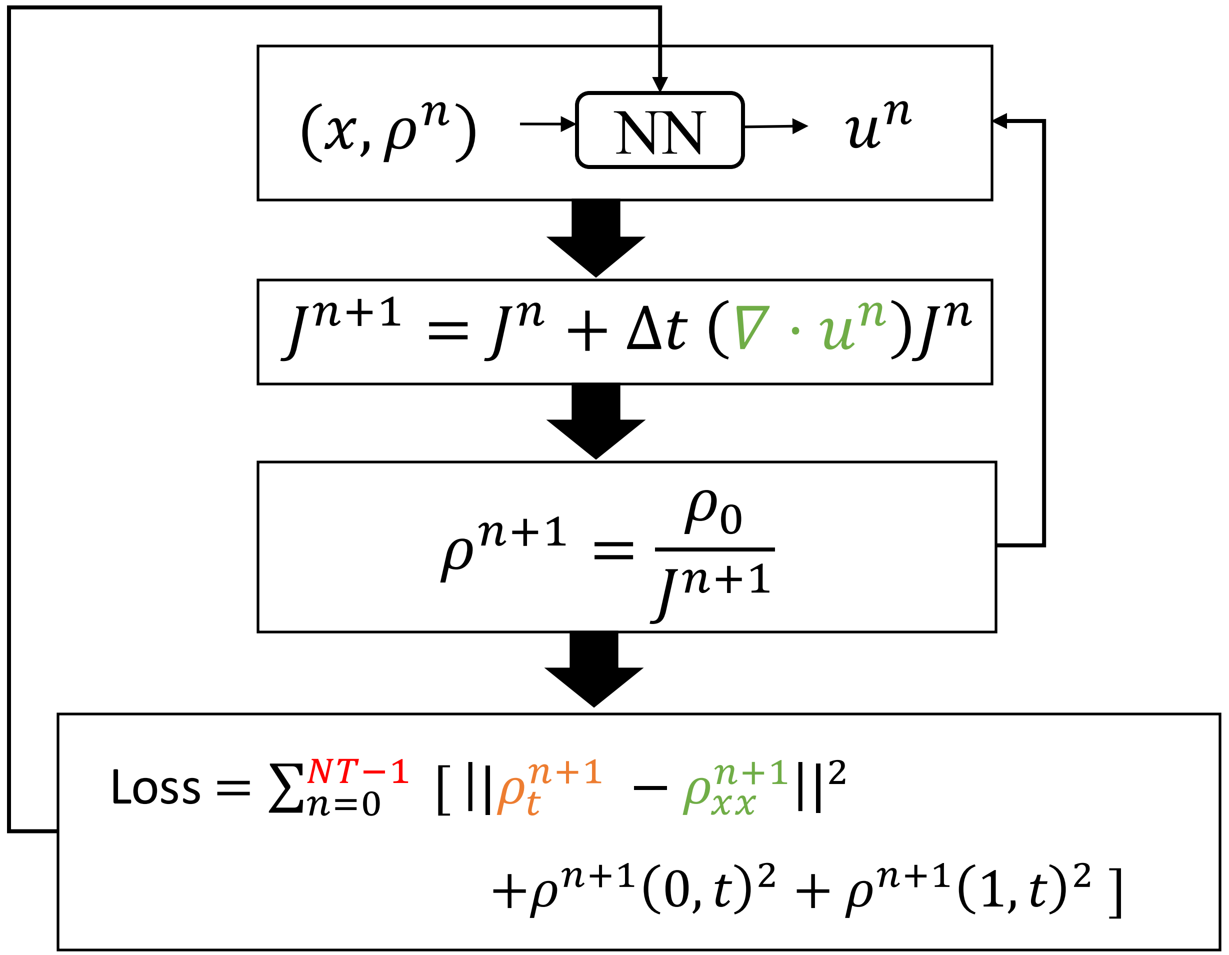
$$\varphi = \frac{\varphi^0}{\det F}. \tag{5}$$



For an $n \times n$ differentiable matrix $A(s)$, we have the following identity:

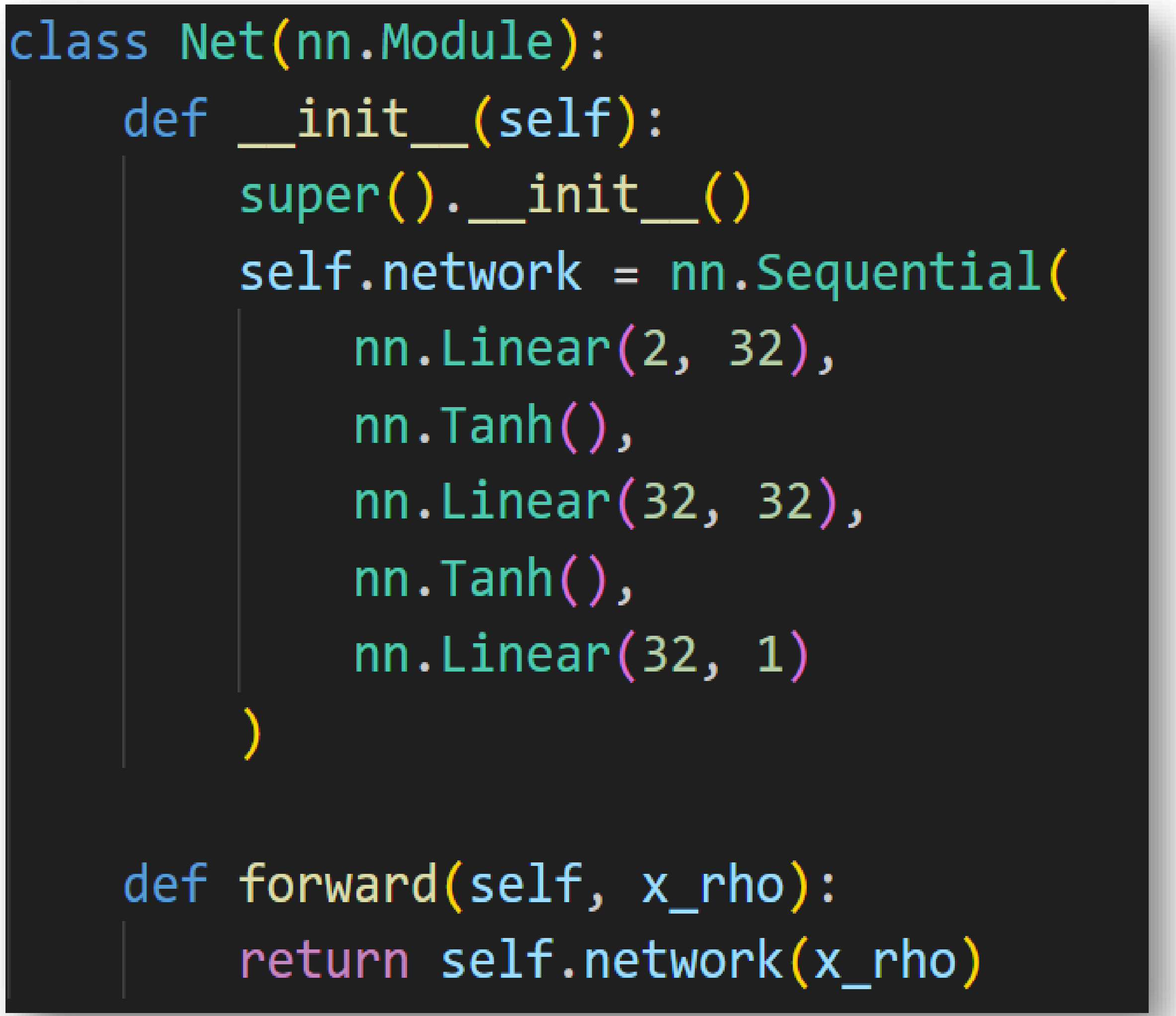
$$\frac{\mathrm{d}}{\mathrm{d}s} \det A(s) = \det A \cdot \operatorname{tr}\left(A^{-1} \frac{\mathrm{d}}{\mathrm{d}s} A\right) = (\nabla \cdot \mathbf{u})\, J$$

Training Pipeline

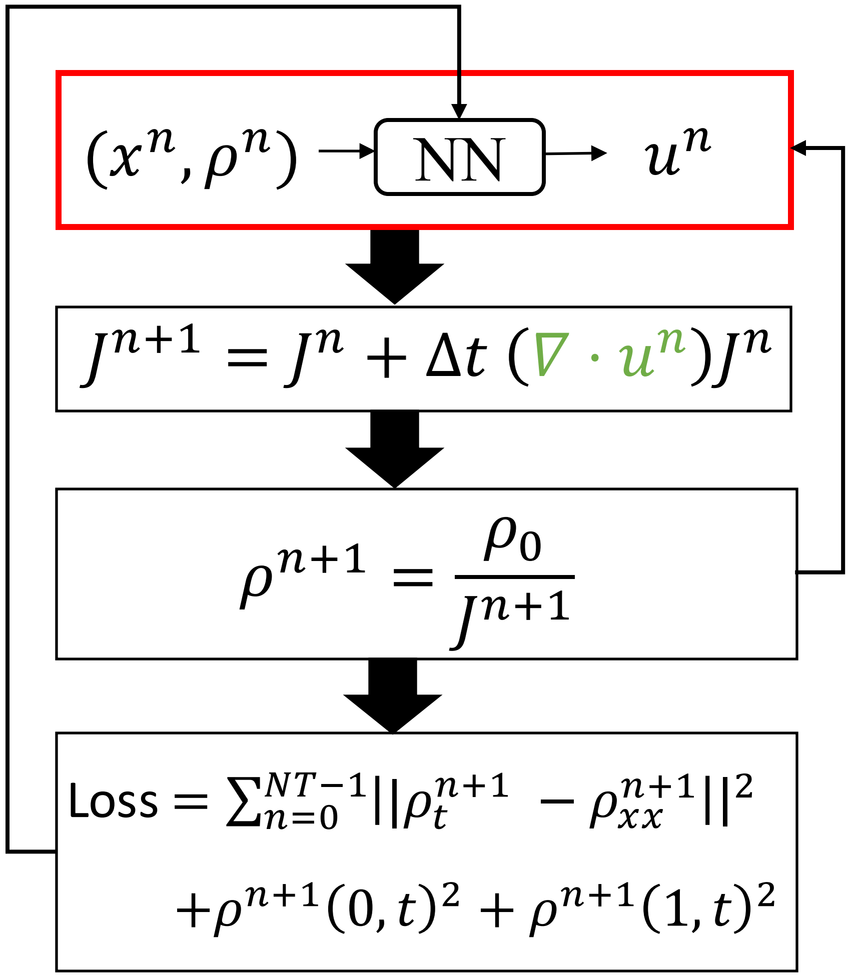

$(x, \rho^n) \rightarrow \boxed{\text{NN}} \rightarrow u^n$
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} [\, ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2 \,]$

🟩 autograd
🟧 Finite difference

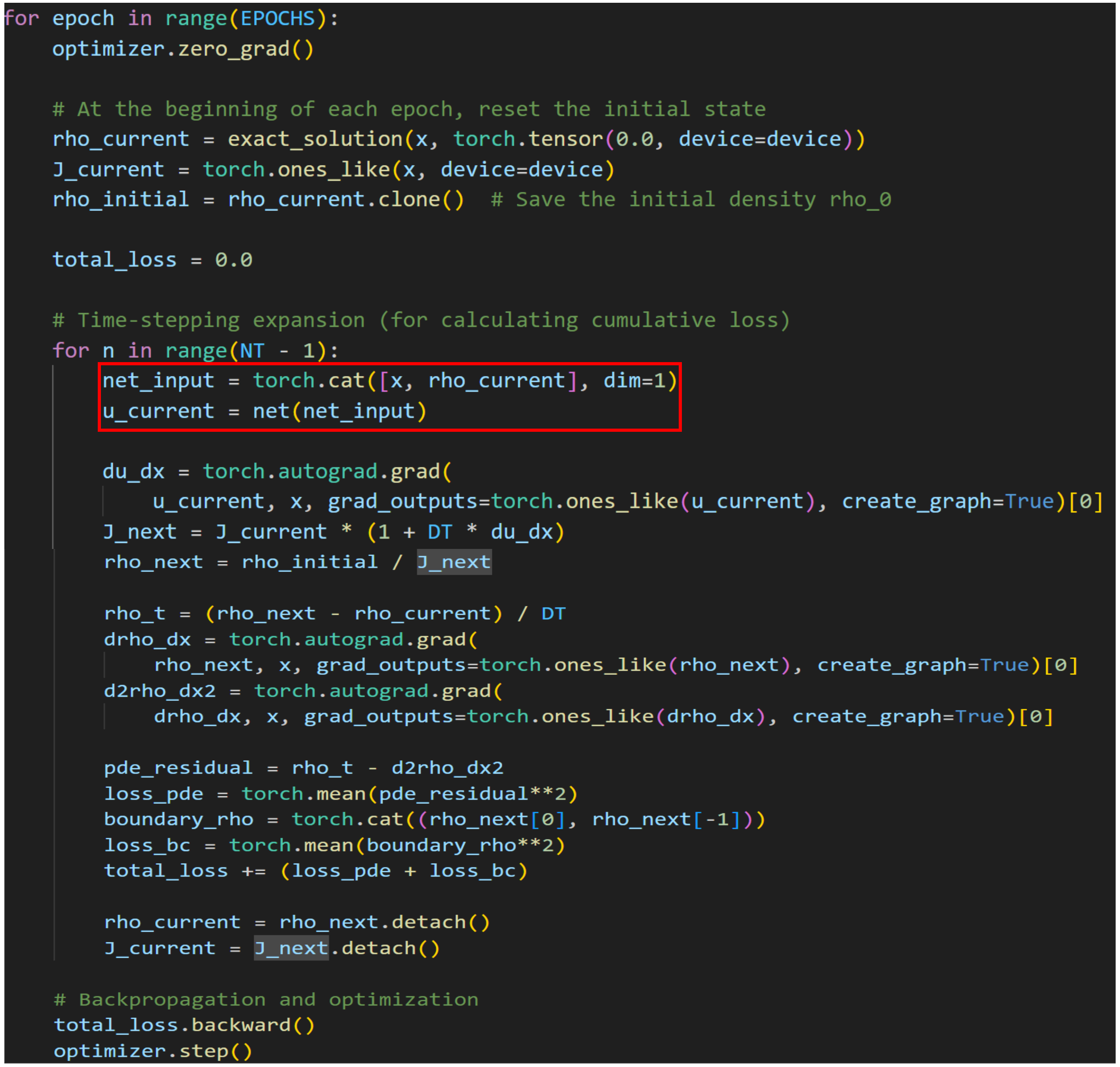
Network



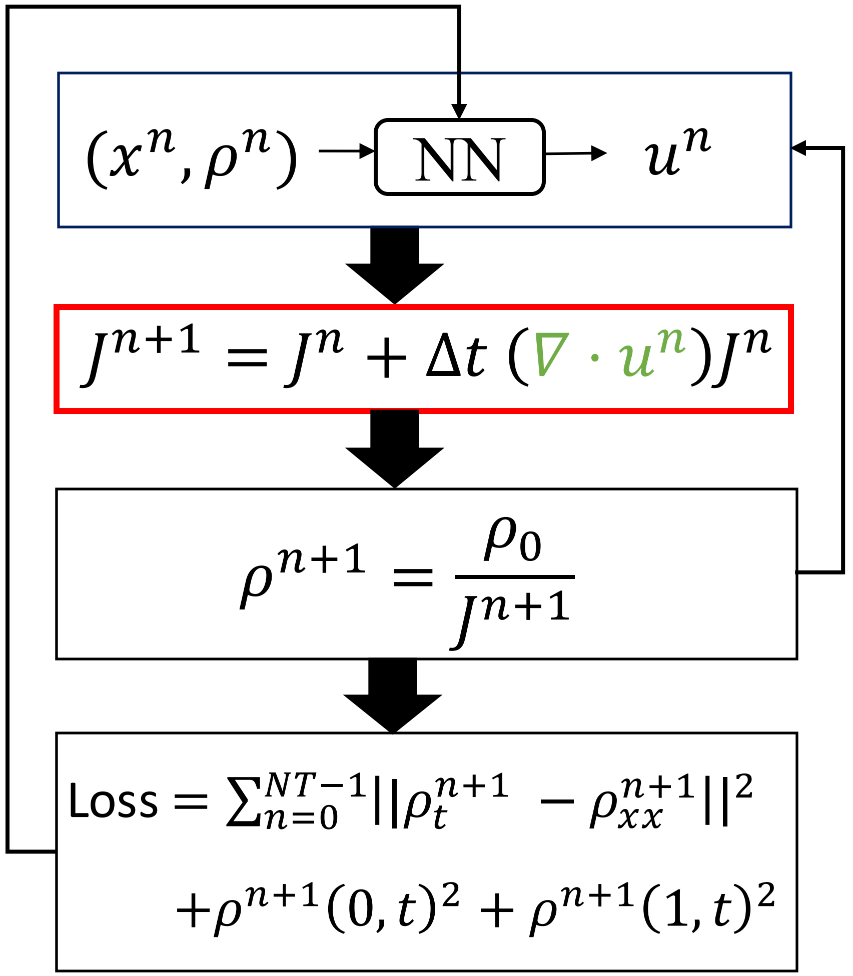
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(2, 32),
            nn.Tanh(),
            nn.Linear(32, 32),
            nn.Tanh(),
            nn.Linear(32, 1)
        )

    def forward(self, x_rho):
        return self.network(x_rho)
```



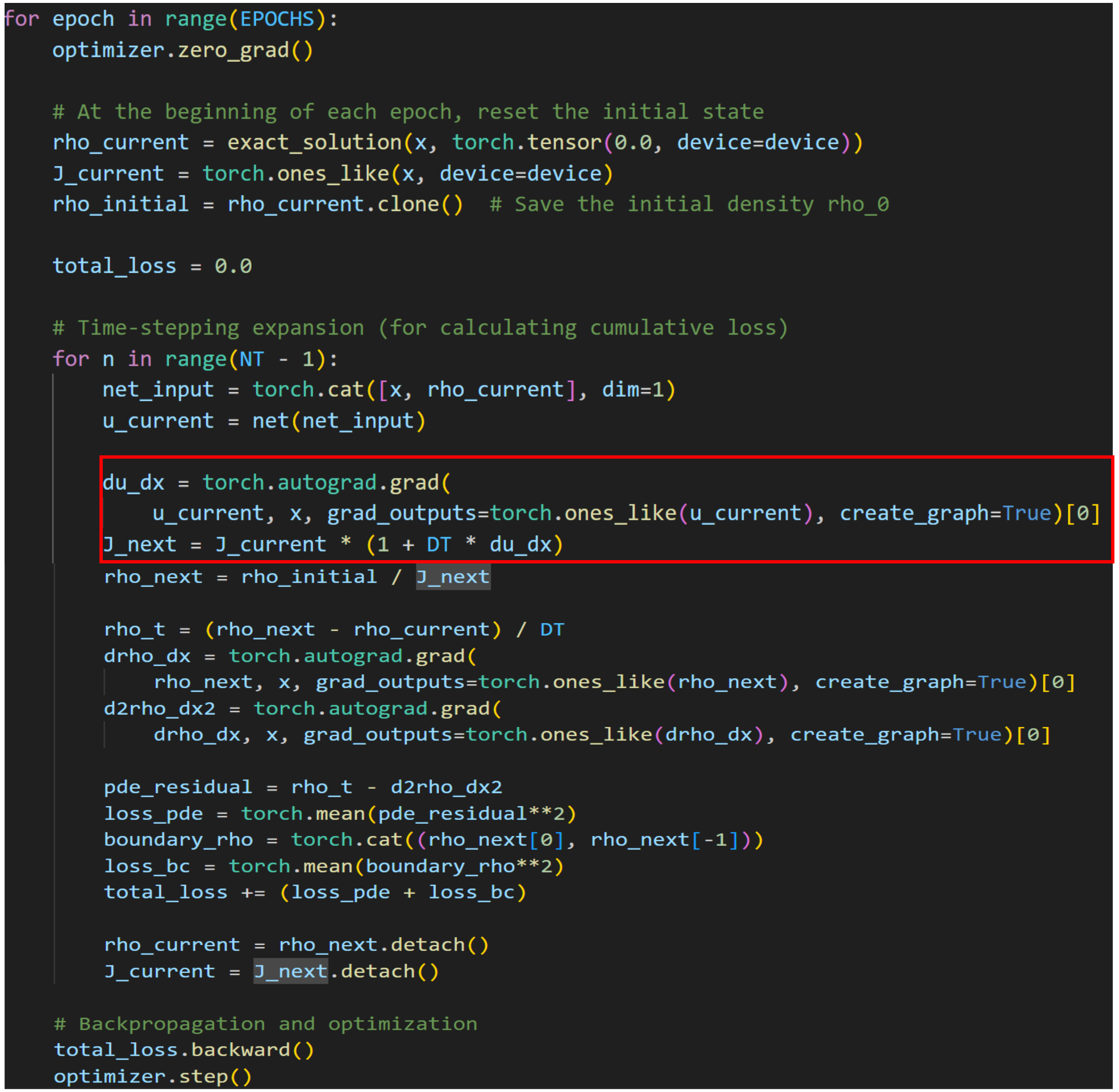
$(x^n, \rho^n) \rightarrow \text{NN} \rightarrow u^n$
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \dfrac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} \|\rho_t^{n+1} - \rho_{xx}^{n+1}\|^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$



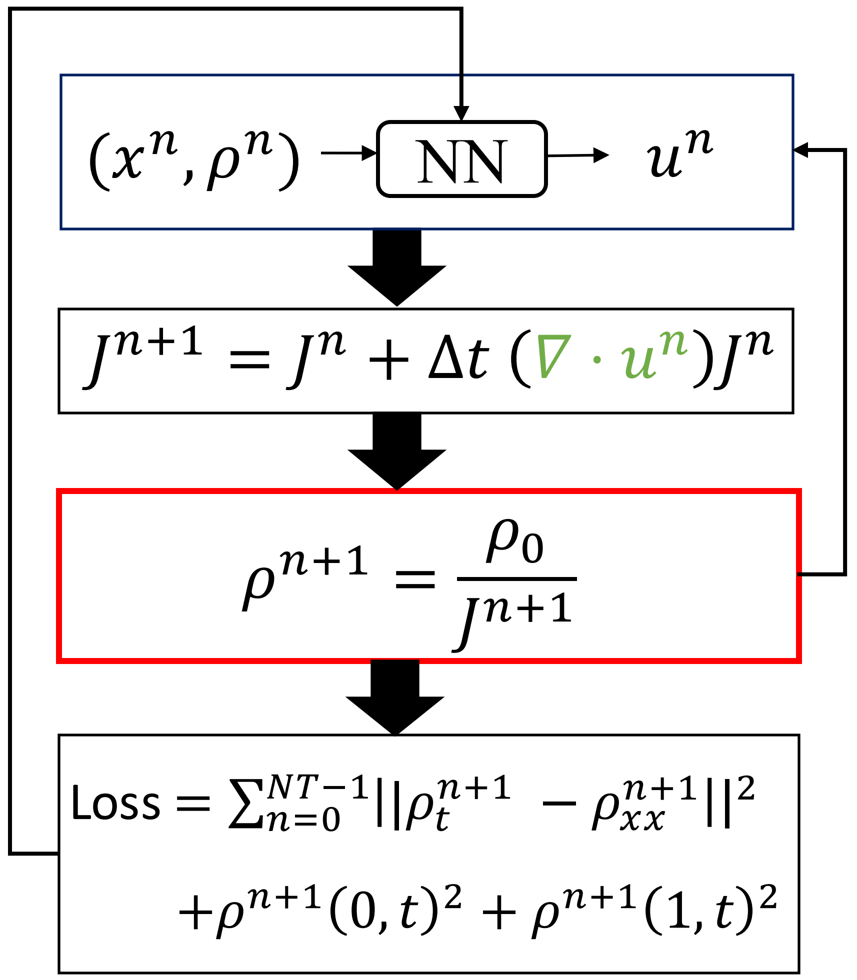
```python
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```

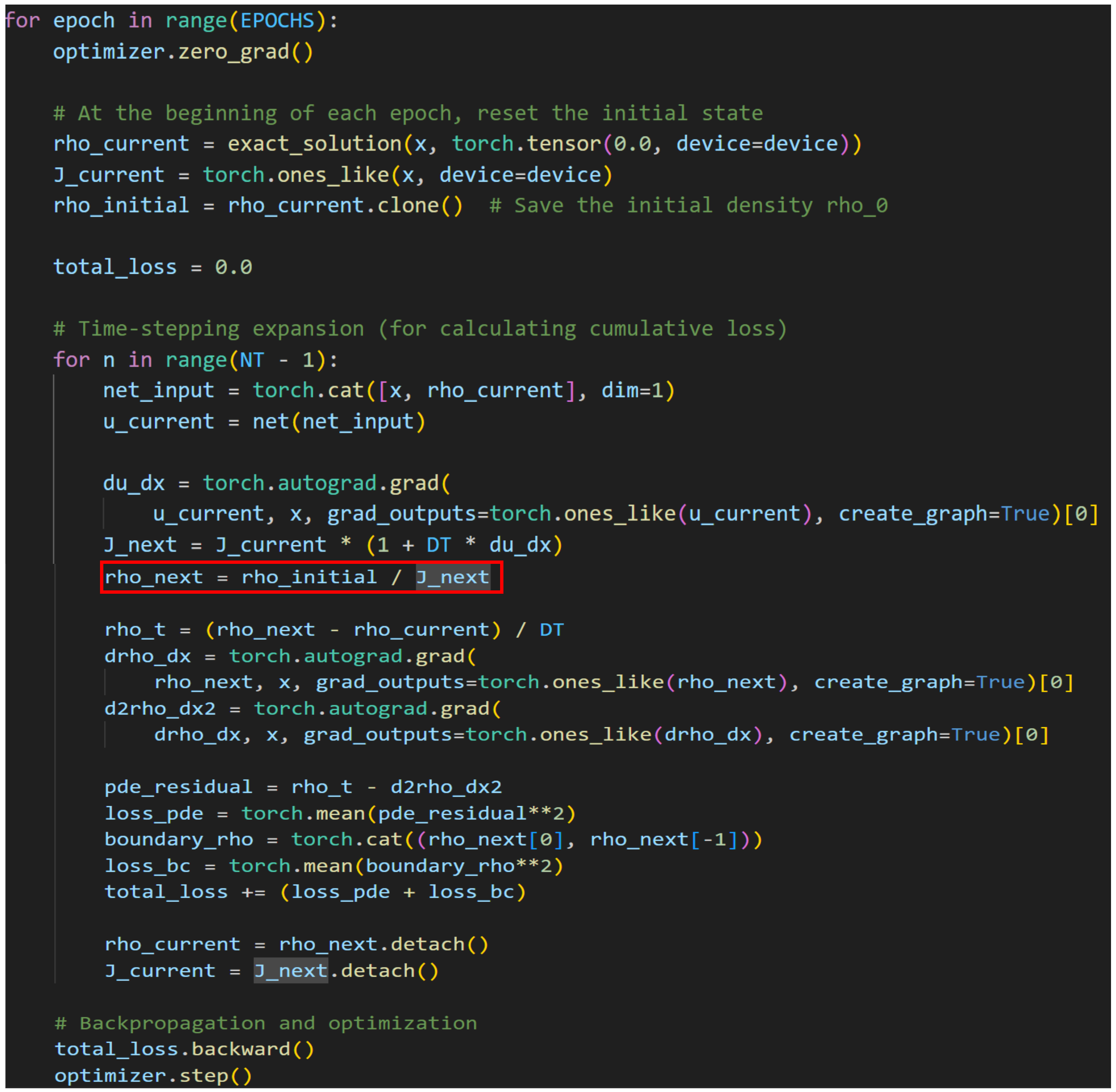


(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$

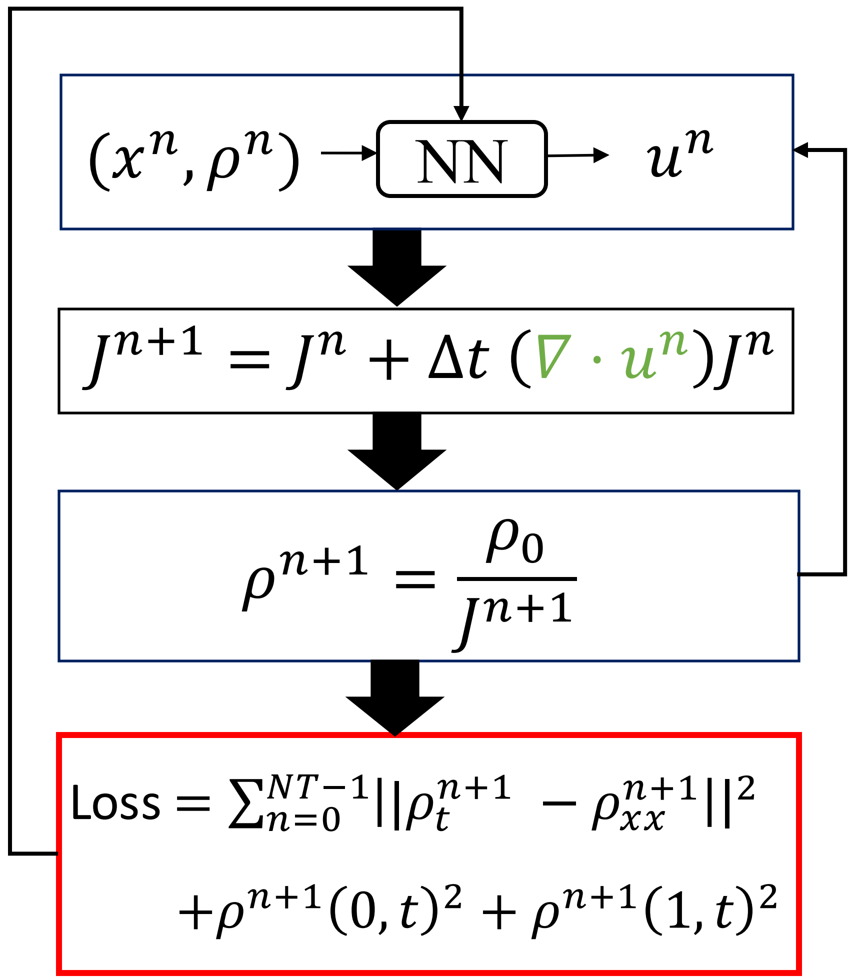


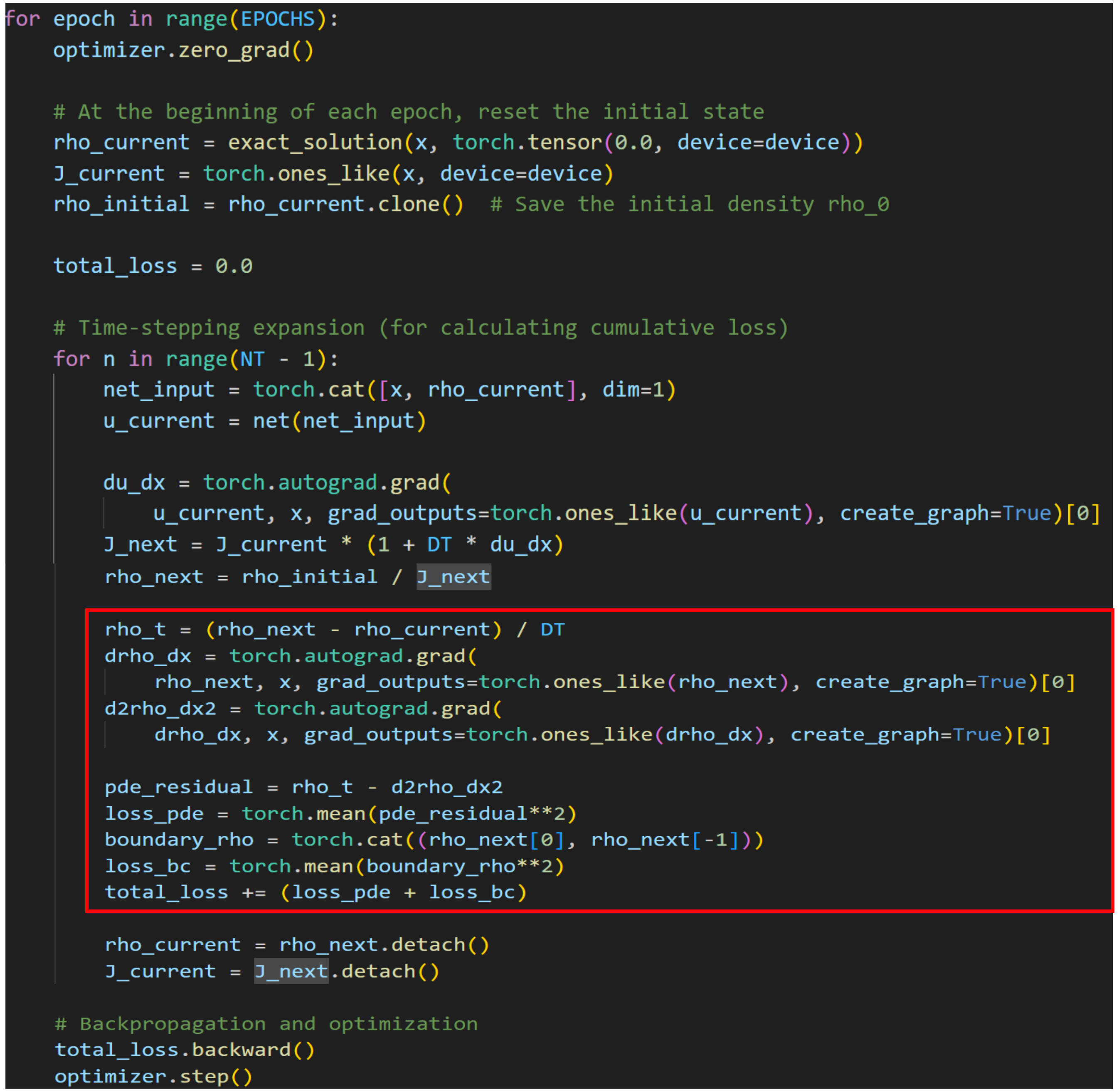
```
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```


(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$

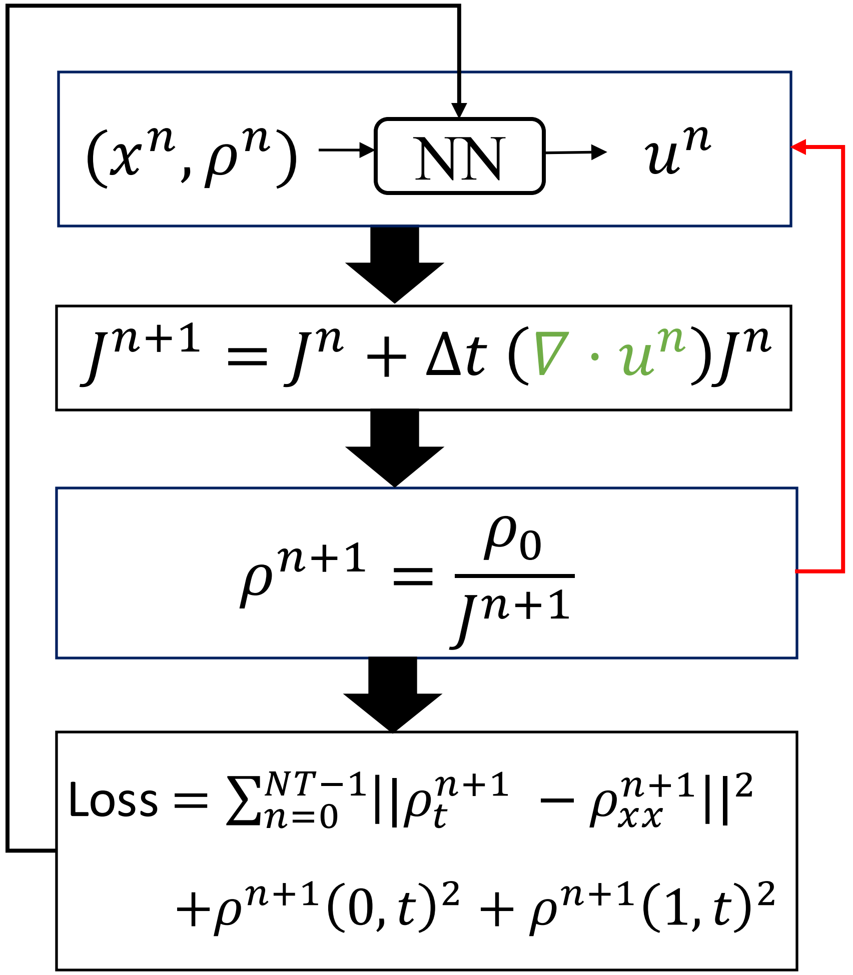


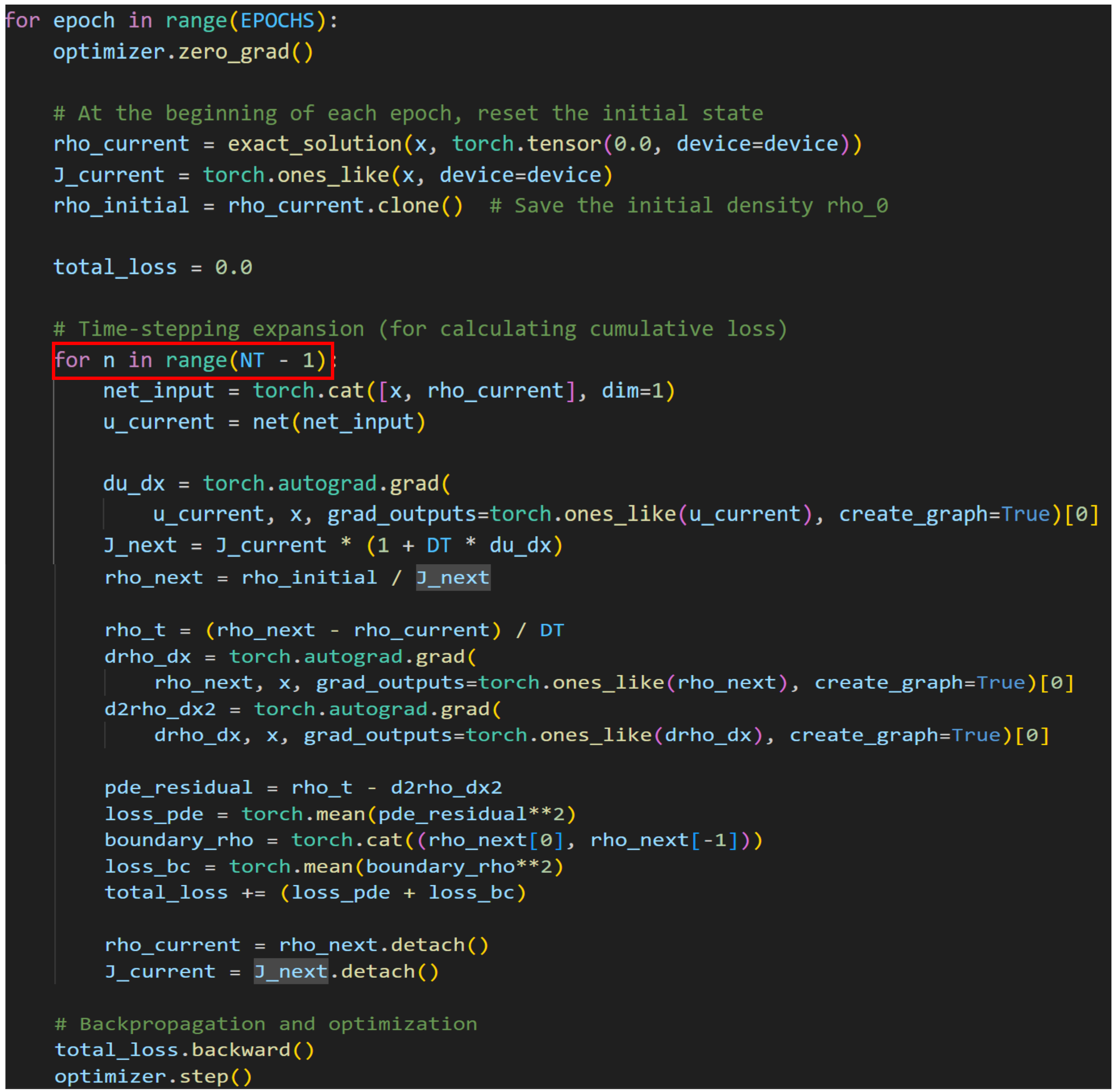
```python
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```


(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \dfrac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2$
$+\rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$

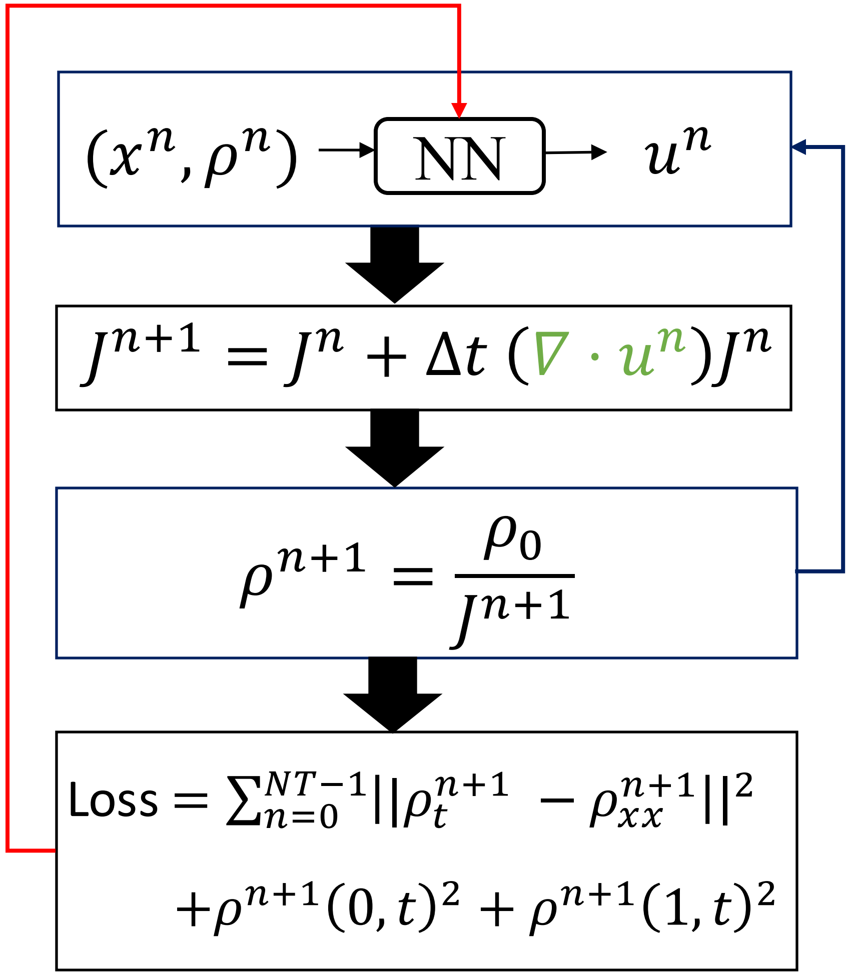


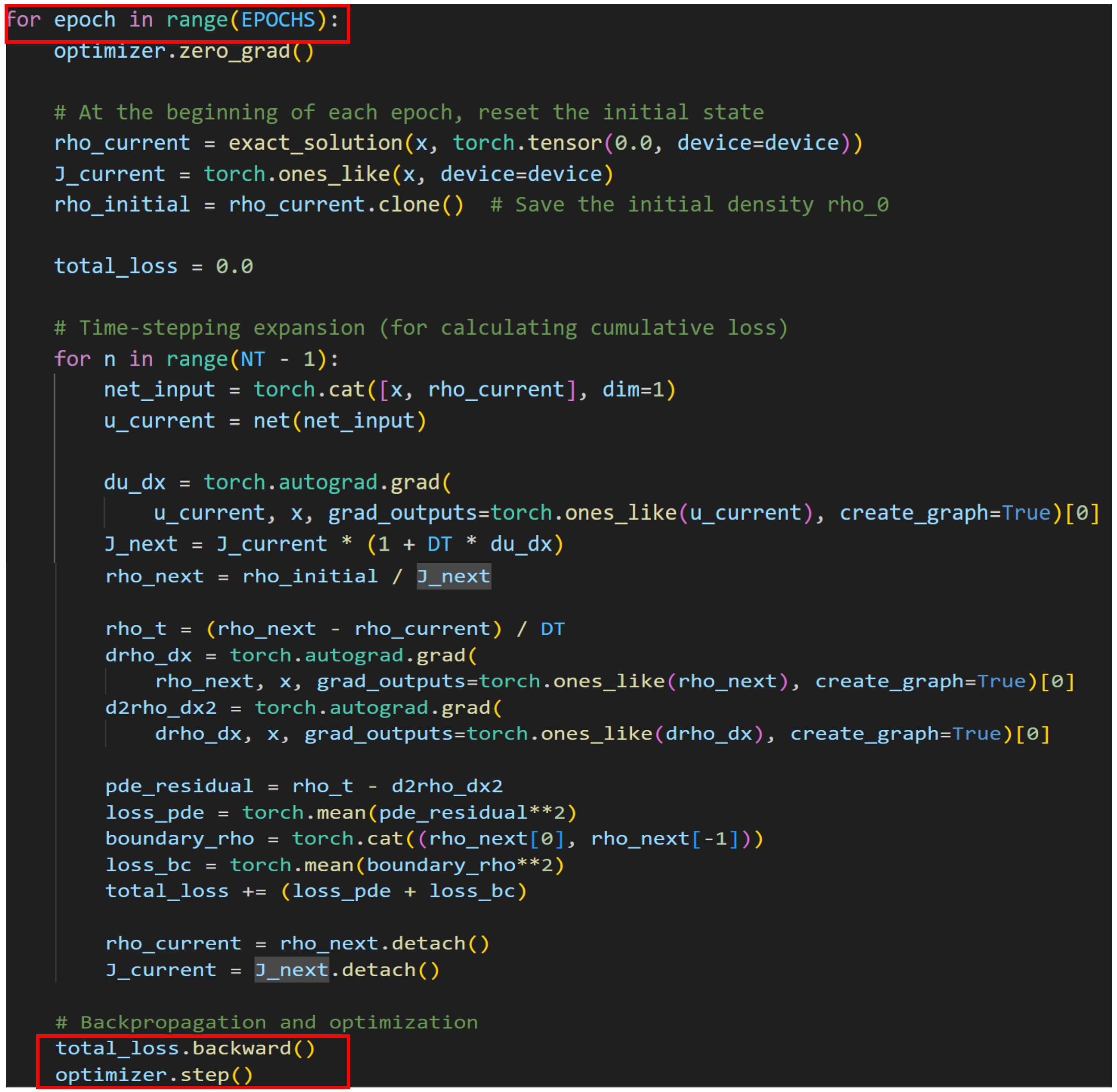
```
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```


(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$



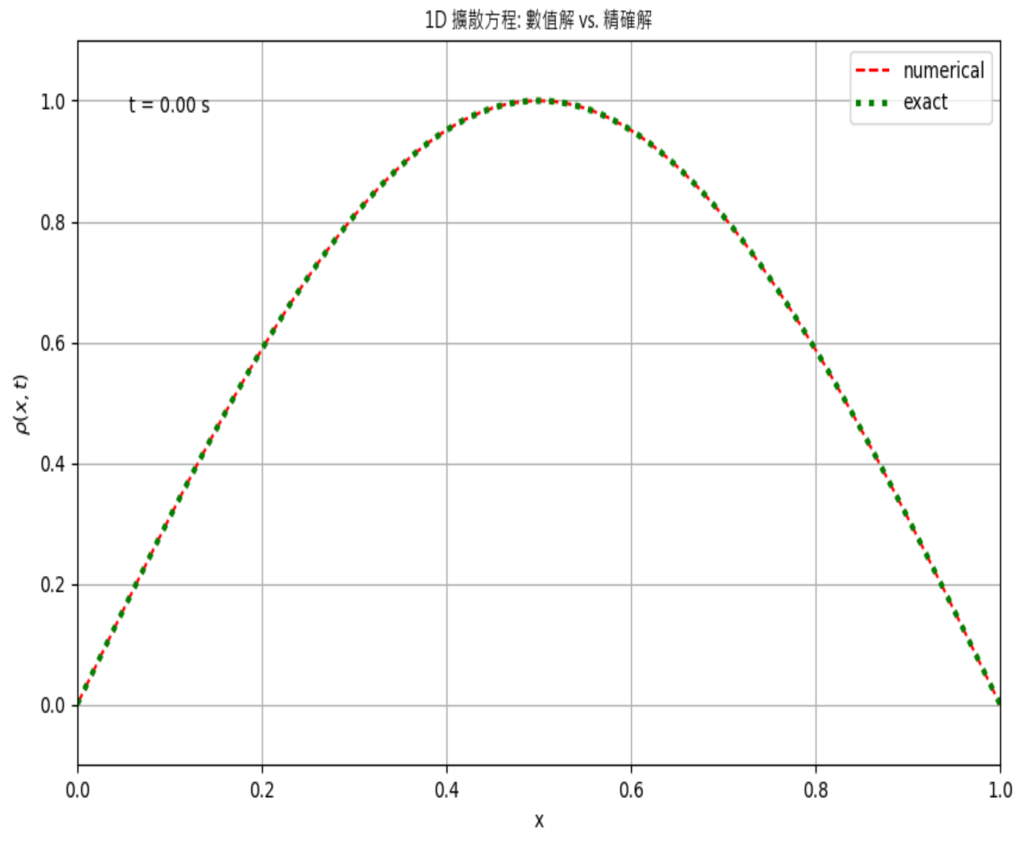
```python
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```

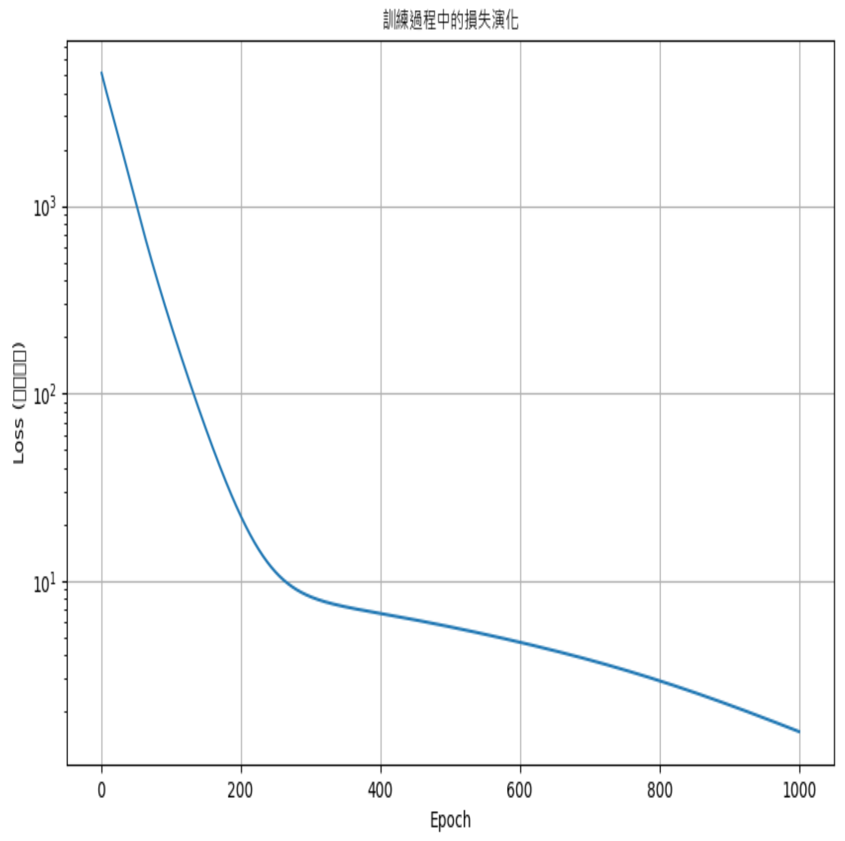

(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho_0}{J^{n+1}}$
$\text{Loss} = \sum_{n=0}^{NT-1} \|\rho_t^{n+1} - \rho_{xx}^{n+1}\|^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2$



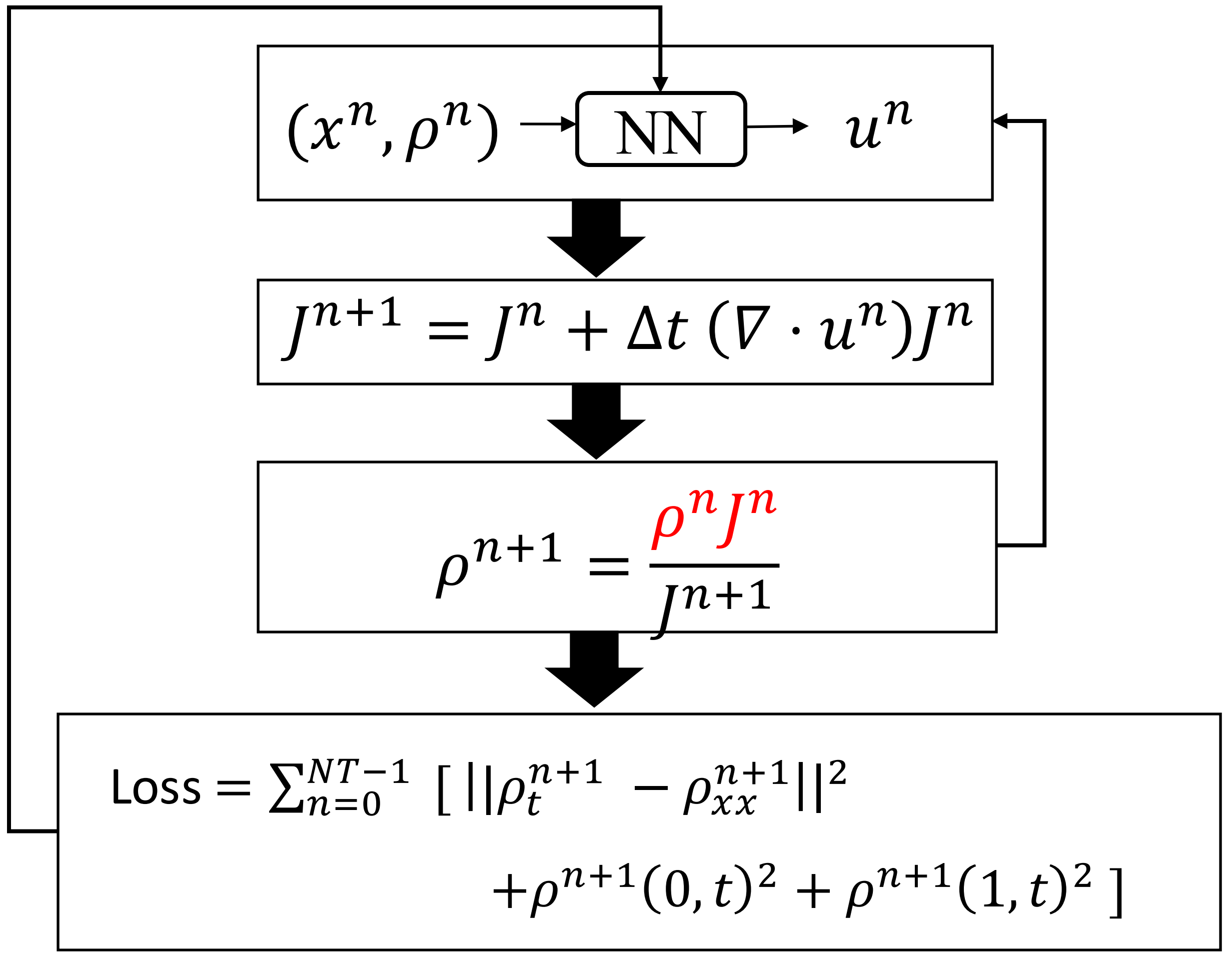
```python
for epoch in range(EPOCHS):
    optimizer.zero_grad()

    # At the beginning of each epoch, reset the initial state
    rho_current = exact_solution(x, torch.tensor(0.0, device=device))
    J_current = torch.ones_like(x, device=device)
    rho_initial = rho_current.clone()  # Save the initial density rho_0

    total_loss = 0.0

    # Time-stepping expansion (for calculating cumulative loss)
    for n in range(NT - 1):
        net_input = torch.cat([x, rho_current], dim=1)
        u_current = net(net_input)

        du_dx = torch.autograd.grad(
            u_current, x, grad_outputs=torch.ones_like(u_current), create_graph=True)[0]
        J_next = J_current * (1 + DT * du_dx)
        rho_next = rho_initial / J_next

        rho_t = (rho_next - rho_current) / DT
        drho_dx = torch.autograd.grad(
            rho_next, x, grad_outputs=torch.ones_like(rho_next), create_graph=True)[0]
        d2rho_dx2 = torch.autograd.grad(
            drho_dx, x, grad_outputs=torch.ones_like(drho_dx), create_graph=True)[0]

        pde_residual = rho_t - d2rho_dx2
        loss_pde = torch.mean(pde_residual**2)
        boundary_rho = torch.cat((rho_next[0], rho_next[-1]))
        loss_bc = torch.mean(boundary_rho**2)
        total_loss += (loss_pde + loss_bc)

        rho_current = rho_next.detach()
        J_current = J_next.detach()

    # Backpropagation and optimization
    total_loss.backward()
    optimizer.step()
```

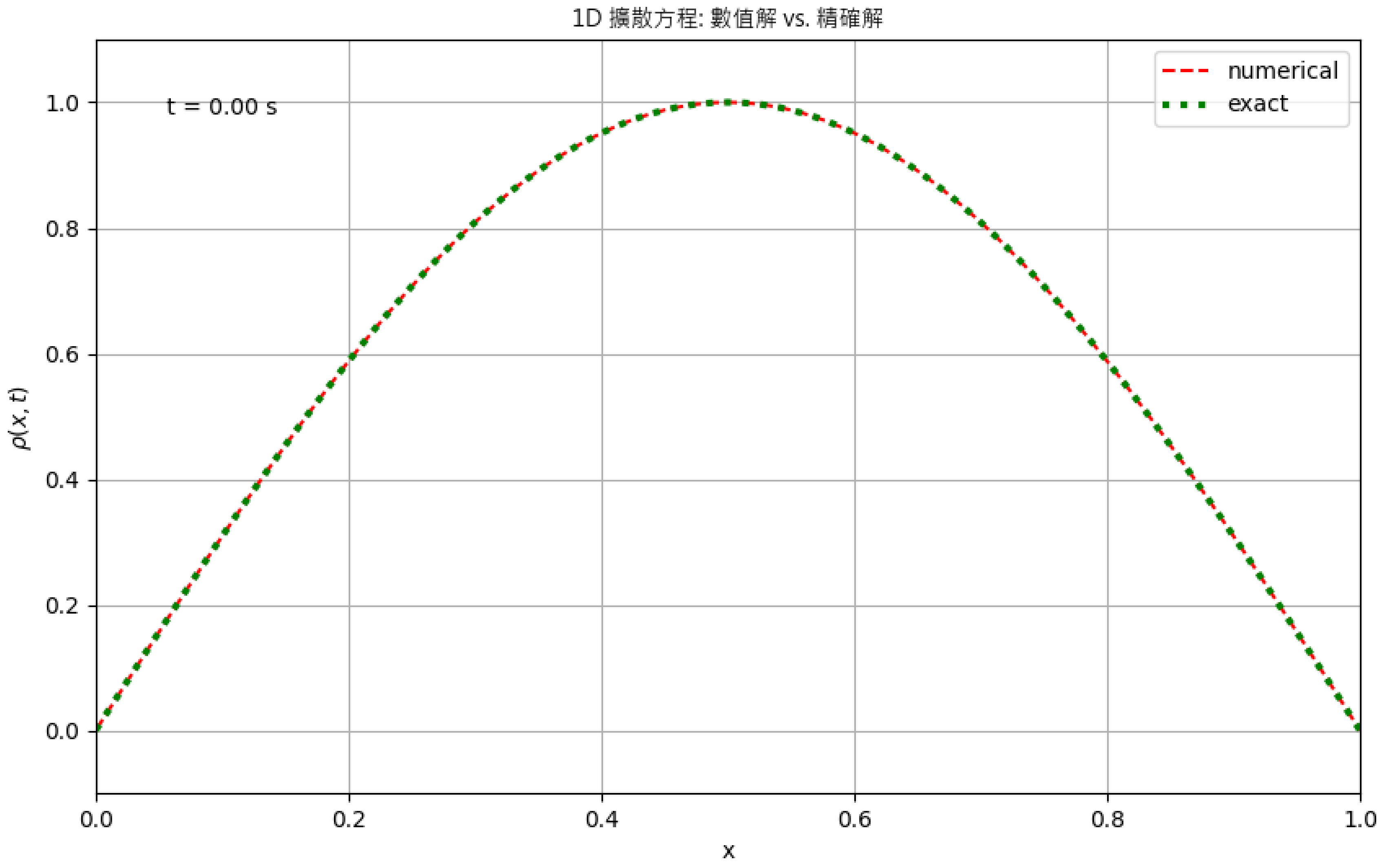
Visualization





Another Experiment


(x^n, ρ^n)
NN
u^n
$J^{n+1} = J^n + \Delta t\,(\nabla \cdot u^n) J^n$
$\rho^{n+1} = \frac{\rho^n J^n}{J^{n+1}}$
Loss $= \sum_{n=0}^{NT-1} \left[\, ||\rho_t^{n+1} - \rho_{xx}^{n+1}||^2 + \rho^{n+1}(0,t)^2 + \rho^{n+1}(1,t)^2 \,\right]$


1D 擴散方程: 數值解 vs. 精確解
t = 0.00 s
numerical
exact
ρ(x, t)
x
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0