PINNs_classes

March 14, 2024

1 PINNs Classes for HW4

[ ]:
```
import torch
import torch.nn as nn
from torch.autograd import grad
import torch.functional as F
import numpy as np


class ffm(nn.Module):
    def __init__(self, in_dim, out_dim, std_dev = 2.0):
        super().__init__()
        self.omega = nn.Parameter(torch.randn(out_dim, in_dim) * std_dev)

    def forward(self, x):
        return torch.cos(F.F.linear(x, self.omega))


class Diffusion_PINNs_1D(nn.Module):
    def __init__(self, in_dim=2, HL_dim=32, out_dim=1, activation=nn.Tanh(),␣
 ↪use_ffm=False, diff_coeff=0.1):
        """
        Parameters
        -------------
        in_dim: the input dimensions - number of independant variables
        HL_dim: the width of the network
        out_dim: the output dimensions - number of dependant variables
        activation: The activation function you wish to use in the network -␣
 ↪the default is nn.Tanh()
        use_ffm: A bool for deciding to use FFM in input or not.
        diff_coeff: The diffusion coefficient used in the PDE
        """
        super().__init__()
        self.diff_coeff = diff_coeff

        # define the network architecture
```

```
        network = [ffm(in_dim, HL_dim)] if use_ffm else [nn.Linear(in_dim,␣
↪HL_dim)]
        network += [
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, out_dim)
                    ]

        # define the network using sequential method
        self.u = nn.Sequential(*network)


    def forward(self, x, t):
        return self.u(torch.cat((x, t), 1))


    def compute_loss(self, x, t, Nx, Nt):
        """
        This is the physics part really
        """
        x.requires_grad=True
        t.requires_grad=True
        u = self.u(torch.cat((x,t), 1))

        # compute PDE derivatives using auto grad
        u_t = grad(u, t, grad_outputs=torch.ones_like(u), create_graph=True)[0]␣
↪# we need to specify the dimension of the output array
        u_x = grad(u, x, grad_outputs=torch.ones_like(u), create_graph=True)[0]
        u_xx = grad(u_x, x, grad_outputs=torch.ones_like(u_x),␣
↪create_graph=True)[0]

        # set a loss function to apply to each of the physics residuals (PDE,␣
↪IC, BC)
        loss_fun = nn.MSELoss()

        # compute the PDE residual loss
        res = u_t - self.diff_coeff * u_xx
        pde_loss = loss_fun(res, torch.zeros_like(res))

        # compute the BC loss
        u_reshaped = u.view(Nx, Nt) # [Nx*Nt, 1] -> [Nx, Nt]
        u_x_reshaped = u_x.view(Nx, Nt) # [Nx*Nt, 1] -> [Nx, Nt]
        bc_loss = loss_fun(u_reshaped[0, :], torch.zeros_like(u_reshaped[0,:]))␣
↪\
```

```
                + loss_fun(u_reshaped[Nx-1, :], torch.
↪zeros_like(u_reshaped[Nx-1,:])) \
                + loss_fun(u_x_reshaped[0, :], u_x_reshaped[Nx-1,:])

        # compute the IC loss
        x_reshaped = x.view(Nx, Nt)
        u_initial = torch.sin(2 * np.pi * x_reshaped[:,0])
        ic_loss = loss_fun(u_initial, u_reshaped[:,0])

        return pde_loss, bc_loss, ic_loss


class Allen_Cahn_1D_PINNs(nn.Module):
    def __init__(self, in_dim=2, HL_dim=64, out_dim=1, activation=nn.Tanh(),␣
↪use_ffm=False):
        """
        Parameters
        -------------
        in_dim: the input dimensions - number of independant variables
        HL_dim: the width of the network
        out_dim: the output dimensions - number of dependant variables
        activation: The activation function you wish to use in the network -␣
↪the default is nn.Tanh()
        """
        super().__init__()

        # define the network architecture
        network = [ffm(in_dim, HL_dim)] if use_ffm else [nn.Linear(in_dim,␣
↪HL_dim)]
        network += [
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, HL_dim), activation,
                    nn.Linear(HL_dim, out_dim)
                    ]

        # define the network using sequential method
        self.u = nn.Sequential(*network)

    def forward(self, x, t):
        return self.u(torch.cat((x, t), 1))


    def compute_loss(self, x, t, Nx, Nt):
        """
        This is the physics part for Allen-Cahn Equation and the ICs/BCs
        """
```

```
        x.requires_grad=True
        t.requires_grad=True
        u = self.u(torch.cat((x,t), 1))

        # compute PDE derivatives using auto grad
        u_t = grad(u, t, grad_outputs=torch.ones_like(u), create_graph=True)[0]␣
↪# we need to specify the dimension of the output array
        u_x = grad(u, x, grad_outputs=torch.ones_like(u), create_graph=True)[0]
        u_xx = grad(u_x, x, grad_outputs=torch.ones_like(u_x),␣
↪create_graph=True)[0]

        # set a loss function to apply to each of the physics residuals (PDE,␣
↪IC, BC)
        loss_fun = nn.MSELoss()

        # compute the PDE residual loss
        res = u_t - 0.0001*u_xx + 5*u**3 -5*u
        pde_loss = loss_fun(res, torch.zeros_like(res))

        # compute the BC loss
        u_reshaped = u.view(Nx, Nt) # [Nx*Nt, 1] -> [Nx, Nt]
        u_x_reshaped = u_x.view(Nx, Nt) # [Nx*Nt, 1] -> [Nx, Nt]
        bc_loss = loss_fun(u_x_reshaped[0, :], u_x_reshaped[Nx-1,:]) \
                + loss_fun(u_reshaped[0, :], u_reshaped[Nx-1,:])

        # compute the IC loss
        x_reshaped = x.view(Nx, Nt)
        u_initial = (x_reshaped[:,0])**2 * torch.cos(np.pi * x_reshaped[:,0])
        ic_loss = loss_fun(u_initial, u_reshaped[:,0])

        return pde_loss, bc_loss, ic_loss
```