```
import logging
import math
import random
import numpy as np
import time
import torch
import torch.nn as nn
from torch import optim
from torch.nn import utils
import matplotlib.pyplot as plt
from torch.optim.lr_scheduler import StepLR
import scipy.io

logger = logging.getLogger(__name__)
PLATFORM_WIDTH = 0.25  # landing platform width
PLATFORM_HEIGHT = 0.06  # landing platform height
FRAME_TIME = 0.1  # time interval
GRAVITY_ACCEL = 0.12  # gravity constant
BOOST_ACCEL = 0.18  # thrust constant
ROTATION_ACCEL = 20  # rotation constant
DRAG_ACCEL = 0.005  # drag constant


class Dynamics(nn.Module):

    def __init__(self):
        super(Dynamics, self).__init__()

    def forward(self, state, action):
        """
        action[0] = thrust controller
        action[1] = omega controller

        state[0] = x
        state[1] = x_dot
        state[2] = y
        state[3] = y_dot
        state[4] = theta
        """
        # Apply gravity
        # Note: Here gravity is used to change velocity which is the second element of the st
        # Normally, we would do x[1] = x[1] + gravity * delta_time
        # but this is not allowed in PyTorch since it overwrites one variable (x[1]) that is
        # Therefore, I define a tensor dx = [0., gravity * delta_time], and do x = x + dx. Th
        delta_state_gravity = torch.tensor([0., 0., 0., -GRAVITY_ACCEL * FRAME_TIME, 0.])

        # Thrust
        # Note: Same reason as above. Need a 5-by-1 tensor.
        N = len(state)
        state_tensor = torch.zeros((N, 5))
```

```
        sin_value = torch.sin(state[:, 4])
        cos_value = torch.cos(state[:, 4])
        state_tensor[:, 0] = -0.5*FRAME_TIME*sin_value
        state_tensor[:, 2] =  0.5*FRAME_TIME*cos_value
        state_tensor[:, 1] =  -sin_value
        state_tensor[:, 3] =  cos_value
        delta_state = BOOST_ACCEL * FRAME_TIME * torch.mul(state_tensor, action[:, 0].reshape

        # Theta
        delta_state_theta = FRAME_TIME * torch.mul(torch.tensor([0., 0., 0., 0, -1.]), action

        state = state + delta_state + delta_state_gravity + delta_state_theta

        # Update state
        step_mat = torch.tensor([[1., FRAME_TIME, 0., 0., 0.],
                                 [0., 1., 0., 0., 0.],
                                 [0., 0., 1., FRAME_TIME, 0.],
                                 [0., 0., 0., 1., 0.],
                                 [0., 0., 0., 0., 1.]])

#         # Noise
#         w = torch.tensor([1,0,0,0,0]) * torch.tensor(np.random.normal(mean, variance, 1))
#         u = torch.tensor([0,1,0,0,0]) * torch.tensor(np.random.normal(mean, variance, 1))
#         t = torch.tensor([0,0,1,0,0]) * torch.tensor(np.random.normal(mean, variance, 1))
#         h = torch.tensor([0,0,0,1,0]) * torch.tensor(np.random.normal(mean, variance, 1))
#         k = torch.tensor([0,0,0,0,1]) * torch.tensor(np.random.normal(mean, variance, 1))
#         noise = w + u

#         if Noise == True:
#             state = torch.matmul(step_mat, state.T) + noise.float()
#         else:

        state = torch.matmul(step_mat, state.T)


        return state.T


class Controller(nn.Module):

    def __init__(self, dim_input, dim_hidden, dim_output):
        """
        dim_input: # of system states
        dim_output: # of actions
        dim_hidden:
        """
        super(Controller, self).__init__()
        # little linear network with ReLU for embeddings
        self.network = nn.Sequential(
            nn.Linear(dim_input, dim_hidden),
            nn.Tanh(),
            nn.Linear(dim_hidden, dim_output),
```

```
            nn.Sigmoid())

    def forward(self, state):
        action = self.network(state)
        return action


class Simulation(nn.Module):

    def __init__(self, controller, dynamics, T, N):
        super(Simulation, self).__init__()
        self.state = self.initialize_state()
        self.controller = controller
        self.dynamics = dynamics
        self.T = T
        self.N = N
        self.theta_trajectory = torch.empty((1, 0))
        self.u_trajectory = torch.empty((1, 0))
#         self.is_Noise = is_Noise

    def forward(self, state):
        self.action_trajectory = []
        self.state_trajectory = []
        for _ in range(T):
            action = self.controller(state)
            state = self.dynamics(state, action)
            self.action_trajectory.append(action)
            self.state_trajectory.append(state)
        return self.error(state)

    @staticmethod
    def initialize_state():
        # state = [1., 0.]  # TODO: need batch of initial states
        state = torch.rand((N, 5))
        state[:, 1] = 0  # vx = 0
        state[:, 3] = 0  # vy = 0
        # TODO: need batch of initial states
        return torch.tensor(state, requires_grad=False).float()

    def error(self, state):
        return torch.mean(state ** 2)


class Optimize:

    # create properties of the class (simulation, parameters, optimizer, lost_list). Where to

    def __init__(self, simulation):
        self.simulation = simulation # define the objective function
        self.parameters = simulation.controller.parameters()
        self.optimizer = optim.LBFGS(self.parameters, lr=0.01) # define the opmization algori
```

```
        self.loss_list = []

    # Define loss calculation method for objective function

    def step(self):
        def closure():
            loss = self.simulation(self.simulation.state)  # calculate the loss of objective
            self.optimizer.zero_grad()
            loss.backward() # calculate the gradient
            return loss

        self.optimizer.step(closure)
        return closure()

    # Define training method for the model

    def train(self, epochs):
        l = np.zeros(epochs)
        for epoch in range(epochs):
            loss = self.step() # use step function to train the model
            self.loss_list.append(loss) # add loss to the loss_list
            print('[%d] loss: %.3f' % (epoch + 1, loss))

            l[epoch]=loss

        plt.plot(list(range(epochs)), l)

        plt.title('Convergence Curve')
        plt.xlabel('Training Iteration')
        plt.ylabel('Error')

        plt.show()

        self.visualize()

    # Define result visualization method

    def visualize(self):
        data = np.array([[self.simulation.state_trajectory[i][N].detach().numpy() for i in ra
        for i in range(self.simulation.N):
            x = data[i, :, 0]  # x position
            y = data[i, :, 2]  # y position
            vx = data[i, :, 1] # Velocity in x direction


        data = np.array([[self.simulation.state_trajectory[i][N].detach().numpy() for i in ra
        for i in range(self.simulation.N):

            plt.plot(vx, y)
        plt.title('Position and Velocity for Rocket Landing')
        plt.xlabel('Rocket Velocity in x direction(m/s)')
        plt.ylabel('y position(m)')
```

```
        plt.show()

        data = np.array([[self.simulation.state_trajectory[i][N].detach().numpy() for i in ra
        for i in range(self.simulation.N):

            plt.plot(range(self.simulation.T), y)
        plt.title('Position and Time for Rocket Landing')
        plt.xlabel('Time (s)')
        plt.ylabel('y position(m)')
        plt.show()


N = 10  # number of samples  for intinal state
T = 100  # number of time steps
dim_input = 5  # state space dimensions
dim_hidden = 6  # latent dimensions
dim_output = 2  # action space dimensions
d = Dynamics()
c = Controller(dim_input, dim_hidden, dim_output)
s = Simulation(c, d, T, N)
o = Optimize(s)
o.train(30)  # training with number of epochs
```

```
[23] loss: 0.010
[24] loss: 0.009
[25] loss: 0.009
[26] loss: 0.009
[27] loss: 0.009
[28] loss: 0.008
[29] loss: 0.008
[30] loss: 0.008
```