Hamiltonian Neural Network

This MATLAB script demonstrates a simple implementation of a Hamiltonian Neural Net-work (HNN) trained using the backpropagation algorithm. Hamiltonian Neural Networks enable the use of Neural Networks under the law of conservation of energy or other invariants. In this code, the task of modeling the dynamics of the frictionless mass-spring system ("Task 1: Ideal Mass-Spring" in Greydanus et al. (2019)) is solved. See this reference for more details.

This script serves as a reference implementation to help users understand how an HNN can be implemented and modify it for more advanced applications.

No MATLAB toolbox is required to run this code, which is particularly useful for educational HNN prototypes or if you want fine-grained control over weight updates, learning rate, activation functions, etc. No dependencies on MATLAB's Deep Learning Toolbox or any other toolboxes exist, therefore it can run on any MATLAB version. It is transparent and easy to extend (ideal for HNN research and learning). The local functions used in this script are listed alphabetically at the end of the main script.

See the file readMe.pdf included in this package for a detailed description.

Note: The results produced by the code may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. For consistency purposes and without loss of generality, the random number generator is appropriately initialized at the be-ginning of the code. If this option is removed, consider running the example a few times and compare the average outcome.

This code is licensed under CC BY 4.0. This license allows reusers to distribute, remix, adapt, and build upon the material in any medium or format, even for commercial purposes. It requires that reusers give credit to the creator.

Contents

Define training data parameters

```
rng(0) % for reproducibility
% Define parameters for data generation
t_span = [0 3];             % start and end times of simulation
timescale = 10;             % refinement of time steps
noise_std = 0.01;           % noise in q, p data
numTrajectories = 50;       % number of independent trajectories
rtol = 1e-10;               % solver tolerance
```

Define training data

```
data=dataGen(t_span,timescale,noise_std,numTrajectories,rtol);
X_all = data(:,1:2);   % Inputs
dq_target = data(:,3);  % target dq/dt
dp_target = data(:,4);  % target dp/dt
```

Define neural network parameters

Define parameters for neural network

```
n_hidden = [50,45];    % Size of hidden layers (two hidden layers)
n_epochs = 10000;      % Number of epochs
n_batch = 750;         % Batch size
learning_rate = 2e-2;  % Learning rate
```

Define the HNN model

Define feedforward backpropagation HNN with two hidden layers

```
hnn = define_hnn(n_hidden);
```

Train the HNN

Train the HNN using manual feedforward–backpropagation

```
[hnn,lossHistory]=train(hnn, X_all,dq_target,dp_target, n_epochs, n_batch,...
    learning_rate);
disp("Training completed.");
```

Plot training results

Plot the loss history

```
figure;
semilogy(lossHistory, 'LineWidth', 2);
xlabel('Epoch');
ylabel('Loss');
title('Hamiltonian NN Training Loss');
grid on;
```

Define testing data

```
data=dataGen(t_span,timescale,noise_std,numTrajectories,rtol);
```

```
X_all = data(:,1:2);   % Inputs
```

Plot the Hamiltonian function value for the testing data

Perform forward pass to calculate the Hamiltonian function value

```
[y,a1,a2] = forwardANN(X_all, hnn.W1, hnn.b1, hnn.W2, hnn.b2, hnn.W3, hnn.b3);
% Calculate the error between the calculated Hamiltonian and its real value
erry=abs(std(y,0)/mean(y));
% Plot the Hamiltonian function (it must be constant)
figure()
plot(y)
xlabel('Observation ID');
ylabel('Hamiltonian function');
title(['Hamiltonian NN output. Error=',num2str(erry)]);
grid on;
axis equal
```

Local function: dataGen

Generate training and testing data for the Hamiltonian Neural Network

```
function data=dataGen(t_span,timescale,noise_std,numTrajectories,rtol)
% Define the symplectic gradient of the Hamiltonian
dynamics_fn = @(t,coords) [  2*coords(2,:);  -2*coords(1,:) ];
% Generate trajectories
t_eval = linspace(t_span(1), t_span(2), ...
    floor(timescale*(t_span(2)-t_span(1)))); % time vector
allData = [];
for s = 1:numTrajectories
    % Random radius and direction
    y0 = rand(2,1)*2 - 1;             % random initial (q,p)
    radius = rand()*0.9 + 0.1;        % radius in [0.1,1.0]
    y0 = y0 / norm(y0) * radius;      % normalize to chosen radius
    % Solve Hamiltonian ODE
    opts = odeset('RelTol',rtol);
    [~, Y] = ode45(@(t,y) dynamics_fn(t,y), t_eval, y0, opts);
    % Extract q, p
    q = Y(:,1);
    p = Y(:,2);
    % Compute dq/dt, dp/dt
    dYdt = zeros(size(Y));
    for i = 1:size(Y,1)
        dqdt = 2*Y(i,2);
        dpdt = -2*Y(i,1);
        dYdt(i,:) = [dqdt, dpdt];
    end
    dqdt = dYdt(:,1);
    dpdt = dYdt(:,2);
    % Add Gaussian noise
    q = q + randn(size(q))*noise_std;
    p = p + randn(size(p))*noise_std;
    % Store results
    allData = [allData; q, p, dqdt, dpdt];
end
% Randomize and trim to 750 samples
idx = randperm(size(allData,1));
data = allData(idx(1:750), :);
```

```
end
```

Local function: define_hnn

Define feedforward Hamiltonian Neural Network

```
function model = define_hnn(n_hidden)
% Initialize weights and biases
model.W1 = 0.1*randn(2, n_hidden(1));
model.b1 = zeros(1, n_hidden(1));
model.W2 = 0.1*randn(n_hidden(1), n_hidden(2));
model.b2 = zeros(1, n_hidden(2));
model.W3 = 0.1*randn(n_hidden(2), 1);
model.b3 = 0;
end
```

Local function: derivativeANN

Compute dy/dx for a 2-hidden-layer ANN (manual backpropagation)

```
function dydx = derivativeANN(x, W1, b1, W2, b2, W3, b3)
%
%   dydx = derivativeANN(x, W1, b1, W2, b2, W3, b3)
%
%   Inputs:
%       x   - input [n x 2]
%       W1  - weights [2 x H]
%       b1  - biases [1 x H]
%       W2  - weights [H x H]
%       b2  - biases [1 x H]
%       W3  - weights [H x 1]
%       b3  - scalar bias
%
%   Output:
%       dydx - [n x 2] = [dy/dx1, dy/dx2]
%
% Forward pass
z1 = x * W1 + b1;       % [1 x H]
h1 = tanh(z1);          % [1 x H]
z2 = h1 * W2 + b2;      % [1 x H]
h2 = tanh(z2);          % [1 x H]
% Local derivatives
dh2dz2 = 1 - h2.^2;     % [1 x H]
dh1dz1 = 1 - h1.^2;     % [1 x H]
% Backpropagation for input derivatives
dYdh2 = W3';                % [1 x H]
dYdz2 = dYdh2 .* dh2dz2;    % [1 x H]
dYdh1 = dYdz2 * W2';        % [1 x H]
dYdz1 = dYdh1 .* dh1dz1;    % [1 x H]
% Gradient wrt inputs
dydx = dYdz1 * W1';         % [1 x 2]
end
```

Local function: forwardANN

Compute forward pass of a 2-hidden-tanh layer ANN.

```
function [y,a1,a2] = forwardANN(x, W1, b1, W2, b2, W3, b3)
%
%   y = forwardANN(x, W1, b1, W2, b2, W3, b3)
%
%   Inputs:
%       x   - input [n x 2]
%       W1  - weights [2 x H]
%       b1  - biases [1 x H]
%       W2  - weights [H x H]
%       b2  - biases [1 x H]
%       W3  - weights [H x 1]
%       b3  - scalar bias
%
%   Output:
%       y   - ANN output(s) [N x 1]
%
% Hidden layer 1
z1 = x * W1 + b1;
a1 = tanh(z1);
% Hidden layer 2
z2 = a1 * W2 + b2;
a2 = tanh(z2);
% Output layer (linear activation)
y = a2 * W3 + b3;
end
```

Local function: hamiltonianGradients

Backpropagation of a Hamiltonian Neural Network. Fully analytical backpropagation using batches of training data

```
function [dW1, db1, dW2, db2, dW3, db3] = hamiltonianGradients(x, dq_target, ...
    dp_target, W1, b1, W2, b2, W3, b3)
% Fully analytical backpropagation, batch version (no loops)
% Each row of x is one input sample
N = size(x,1);
% Forward
z1 = x * W1 + b1;          % N x H
a1 = tanh(z1);             % N x H
z2 = a1 * W2 + b2;         % N x H
a2 = tanh(z2);             % N x H
s1 = 1 - a1.^2;            % N x H
s2 = 1 - a2.^2;            % N x H
alpha = (W3') .* s2;       % N x H
beta = alpha * W2';        % N x H
v = beta .* s1;            % N x H
g = v * W1';               % N x 2
g1 = g(:,1);
g2 = g(:,2);
eq = g2 - dq_target;       % N x 1
ep = g1 + dp_target;       % N x 1
D = [2*ep, 2*eq];          % N x 2
% Backpropagation
r = D * W1;                % N x H
dL_dbeta = r .* s1;        % N x H
dL_dalpha = dL_dbeta * W2; % N x H
dL_ds2 = dL_dalpha .* (W3)'; % N x H
dL_dz2 = dL_ds2 .* (-2 .* a2 .* s2); % N x H
dL_da1_from_z2 = dL_dz2 * W2';       % N x H
dL_dz1_from_z2 = dL_da1_from_z2 .* s1; % N x H
```

```
dL_ds1 = r .* beta;        % N x H
dL_dz1_from_s1 = dL_ds1 .* (-2 .* a1 .* s1); % N x H
dL_dz1 = dL_dz1_from_z2 + dL_dz1_from_s1;    % N x H
% Parameter gradients (averaged)
dW3 = mean(s2 .* dL_dalpha,1)';         % H x 1
db3 = zeros(1,1);                   % scalar (∂L/∂b3 = 0)
dW2 = ((a1' * dL_dz2) + (dL_dbeta' * alpha)) / N;  % H x H
db2 = mean(dL_dz2,1);               % 1 x H
dW1_direct = (D') * v / N;          % 2 x H
dW1_from_z1 = (x') * dL_dz1 / N;    % 2 x H
dW1 = dW1_from_z1 + dW1_direct;     % 2 x H
db1 = mean(dL_dz1,1);               % 1 x H
end
```

Local function: train

Train a Hamiltonian Neural Network

```
function [hnn,lossHistory]=train(hnn, X_all, dq_target, dp_target, n_epochs,...
    n_batch, learning_rate)
Ntot = size(X_all,1);
lossHistory = zeros(n_epochs,1);
iter = 0;
for epoch = 1:n_epochs
    perm = randperm(Ntot);
    for i0 = 1:n_batch:Ntot
        iter = iter + 1;
        idx = perm(i0:min(i0+n_batch-1,Ntot));
        X_batch = X_all(idx,:);
        dq_target_batch = dq_target(idx);
        dp_target_batch = dp_target(idx);
        % Forward: compute predicted derivatives
        dHdX = derivativeANN(X_batch, hnn.W1, hnn.b1, hnn.W2, hnn.b2,...
            hnn.W3, hnn.b3);
        dHdq = dHdX(:,1);
        dHdp = dHdX(:,2);
        dq_pred = dHdp;
        dp_pred = -dHdq;
        % Compute loss
        loss_dq = (dq_pred - dq_target_batch).^2;
        loss_dp = (dp_pred - dp_target_batch).^2;
        loss = mean(loss_dq + loss_dp);
        lossHistory(epoch) = loss;
        % Backpropagation: parameter gradients
        [dW1, db1, dW2, db2, dW3, db3] = ...
            hamiltonianGradients(X_batch, dq_target_batch, dp_target_batch, ...
            hnn.W1, hnn.b1, hnn.W2, hnn.b2, hnn.W3, hnn.b3);
        % Update parameters
        hnn.W1 = hnn.W1 - learning_rate * dW1;
        hnn.b1 = hnn.b1 - learning_rate * db1;
        hnn.W2 = hnn.W2 - learning_rate * dW2;
        hnn.b2 = hnn.b2 - learning_rate * db2;
        hnn.W3 = hnn.W3 - learning_rate * dW3;
        hnn.b3 = hnn.b3 - learning_rate * db3; % db3 is zero but keep for clarity
    end
    % Display progress occasionally
    if mod(epoch, 500) == 0
        fprintf("Epoch %d | Loss: %.6f\n", epoch, loss);
    end
```

```
end
end
```