Hamiltonian Neural Networks

Anuj Jagannath Said J Jayagowtham
Praveen Shivsurya

Indian Institute of Technology Madras

May 21, 2025

Theory - Hamiltonian Mechanics (Recap...)

- **Hamiltonian mechanics** describes the dynamics of systems using total energy $H = T + V$, where energy is conserved over time in conservative, force-free systems.

Core Idea in HNN's

The change in momenta and the change in coordinates with respect to time is given by the following equations

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i}, \quad \frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i}$$

These equations describe the time evolution of generalized coordinates and momenta which are used to predict (q_i^{t+1}, p_i^{t+1}) from (q_i^t, p_i^t) via

$$(q_i^{t+1}, p_i^{t+1}) = (q_i^t, p_i^t) + \int_t^{t+1} S(q, p)\, dt$$

Models - Hamiltonian Neural Network (Recap...)

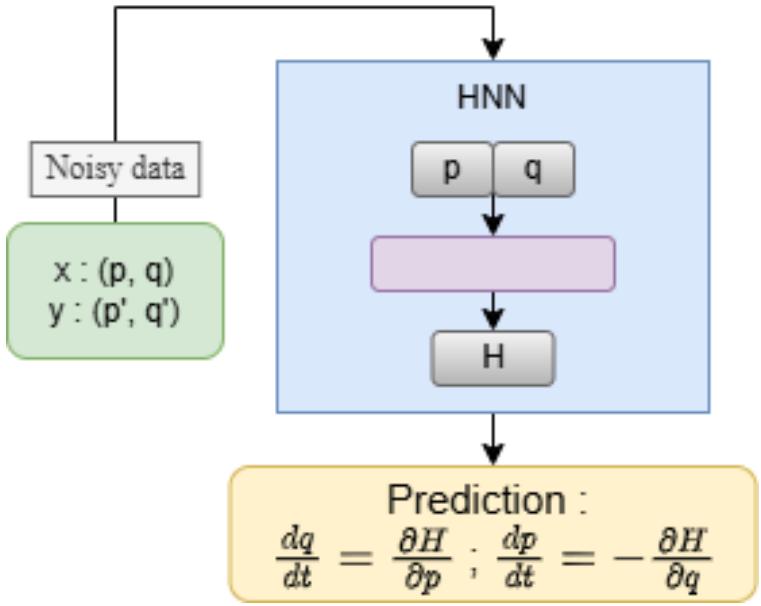


Figure: HNN - training process where for given set of (p, q) model predicts $\frac{dp_i}{dt}, \frac{dq_i}{dt}$

Loss function used to train HNN uses:

$$\mathcal{L}_{hnn} = \left\| \frac{\partial H}{\partial q_t} + \dot{p}_t \right\|_2^2 + \left\| \frac{\partial H}{\partial p_t} - \dot{q}_t \right\|_2^2 \tag{1}$$

Models - LASSO Regression (Recap...)

Main Theme Using polynomial features, we construct a functional form for the Hamiltonian. Then, by applying **LASSO** regression — which promotes sparsity — we identify the most important contributing terms in the Hamiltonian.

1. **Feature Matrices:**

$$y_1 = \frac{dq}{dt} = X_1 W \quad \text{and} \quad y_2 = \frac{dp}{dt} = X_2 W,$$

where : W is the vectorized form of the coefficients $\{\alpha_{ij}\}$.

X_1 and X_2 are calculated by differenting the function $\hat{H}(q, p)$ wrt the inputs accordingly

2. **Regression** By concatenating the two systems, we obtain an equivalent LS problem:

$$\min_W \left\| \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} W - \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right\|_2^2 + \lambda \|W\|_1.$$

Regularization in HNN

Regularisers which we used in Hamiltonian Neural networks are:

1. **Dropout**: Randomly drops a fraction of the neurons during training. It kind of train several neural networks having different architectures by creating multiple instances of the same network.
2. L_2 **Regularization (Weight Decay)**: We add a penalty term $\lambda \| W \|_2$ to the loss function, where W are the model weights and λ controls the strength of the penalty. This discourages large weights, promotes smoother solutions, and helps avoid overfitting.
3. **Early Stopping**: Monitors the validation loss during training and stops the training process when performance on the validation set no longer improves, preventing overfitting to the training data.

Incorporating Trajectory-Based Losses in HNN

We explored two ways to inject trajectory information into our Hamiltonian Neural Network, yielding two losses. The loss function is given by

$$\mathcal{L}_{\text{traj}} = \sum_{j=1}^{M} \sum_{t=1}^{N} \left(\| p_{j,t}^{\text{act}} - p_{j,t}^{\text{pred}} \|_2^2 + \| q_{j,t}^{\text{act}} - q_{j,t}^{\text{pred}} \|_2^2 \right).$$

1. **Full-Trajectory Loss.** From the initial state $(p_{j,0}, q_{j,0})$ of each of the M trajectories, we let the model predict the entire trajectory allowing the errors to propagate. We then penalize deviations over the entire trajectory.

$$p_{j,t}^{\text{pred}}, q_{j,t}^{\text{pred}} = f(p_{j,0}, q_{j,0}, p_{j,t-1}^{\text{pred}}, q_{j,t-1}^{\text{pred}})$$

2. **One-Step Prediction Loss.** At each time step t, we let the model predict only the next states using the true state at that time instant.

$$p_{j,t}^{\text{pred}}, q_{j,t}^{\text{pred}} = f(p_{j,0}, q_{j,0}, p_{j,t-1}^{\text{act}}, q_{j,t-1}^{\text{act}})$$

Incorporating Trajectory-Based Losses in HNN (contd..)

Finally, we combine these with the standard HNN loss and an L_2 weight-decay term, we get,

Final Loss

Using the previous slide results,

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{HNN}} \; + \; \alpha \, \mathcal{L}_{\text{traj}} \; + \; \lambda \, \|W\|_2^2,$$

where,

- $\mathcal{L}_{\text{traj}}$ is either $\mathcal{L}_{\text{full}}$ or $\mathcal{L}_{\text{step}}$
- α balances the trajectory loss contribution
- and λ controls the weight-decay strength.

Methodology

Full trajectory algorithm

1. For $(\hat{p}_i, \hat{q}_i)$, using the model predict $(\hat{\dot{p}}_i, \hat{\dot{q}}_i)$
2. Using RK4 scheme, numerically integrate the dynamics function to get $(\hat{p_{i+1}}, \hat{q_{i+1}})$
3. Repeat this process till the end of trajectory.
4. Make the loss at end of all trajectories 0, since we don't have a next point in the trajectory.

One step loss algorithm

1. For (p_i, q_i), using the model predict $(\hat{\dot{p}}_i, \hat{\dot{q}}_i)$
2. Using RK4 scheme, numerically integrate the dynamics function to get $(\hat{p_{i+1}}, \hat{q_{i+1}})$
3. Calculate the one-step loss and
4. Using torch functionalities, simultaneously compute the loss for all points in the trajectory.
5. Make the loss at end of all trajectories 0, since we don't have a next point in the trajectory.

Code blocks

Following are few snippets of code augmented with the existing HNN code.

```
next_pts = get_next_point_with_model(model_fwd, x, small_t_eval)
next_pts[traj_len-1:-1:traj_len] = x[traj_len::traj_len]  #prevent penalization at end of trajectory
extra_loss = L2_loss(x[1:], next_pts[:-1])
loss += alpha*extra_loss
```

Figure: Loss addition

```
def get_next_point_with_model(model_fwd, y0, dt):
    t0 = 0.0
    dy = rk4(model_fwd, y0, t0, dt)   # shape: (B, 2)
    return y0 + dy
```

Figure: Numerical integration

```
def rk4(fun, y0, t, dt, *args, **kwargs):
  dt2 = dt / 2.0
  k1 = fun(y0, t, *args, **kwargs)
  k2 = fun(y0 + dt2 * k1, t + dt2, *args, **kwargs)
  k3 = fun(y0 + dt2 * k2, t + dt2, *args, **kwargs)
  k4 = fun(y0 + dt * k3, t + dt, *args, **kwargs)
  dy = dt / 6.0 * (k1 + 2 * k2 + 2 * k3 + k4)
  return dy
```

Figure: Runge Kutta 4 scheme

Full trajectory loss

Sample results with full trajectory loss. We didn't pursue this owing to the high compute power requirements.

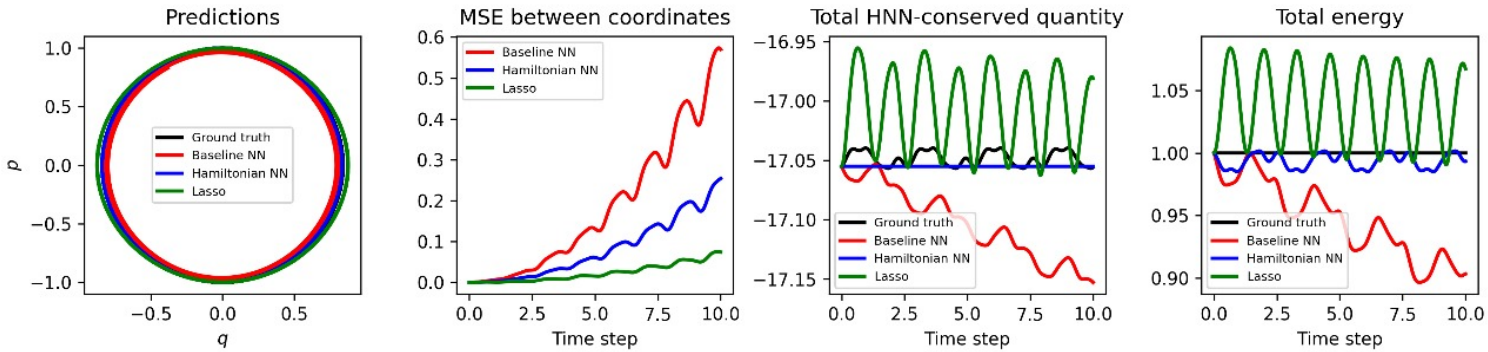


Figure: Before improvement

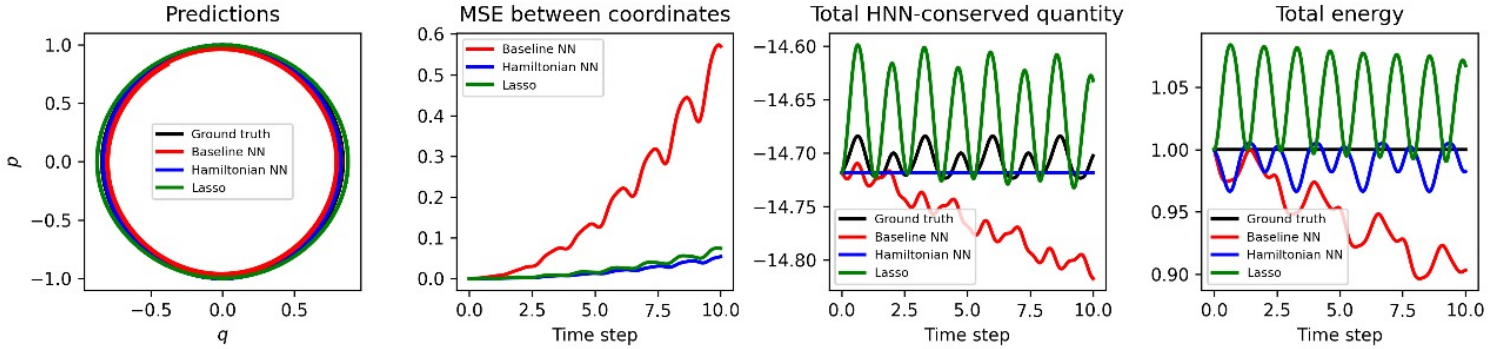
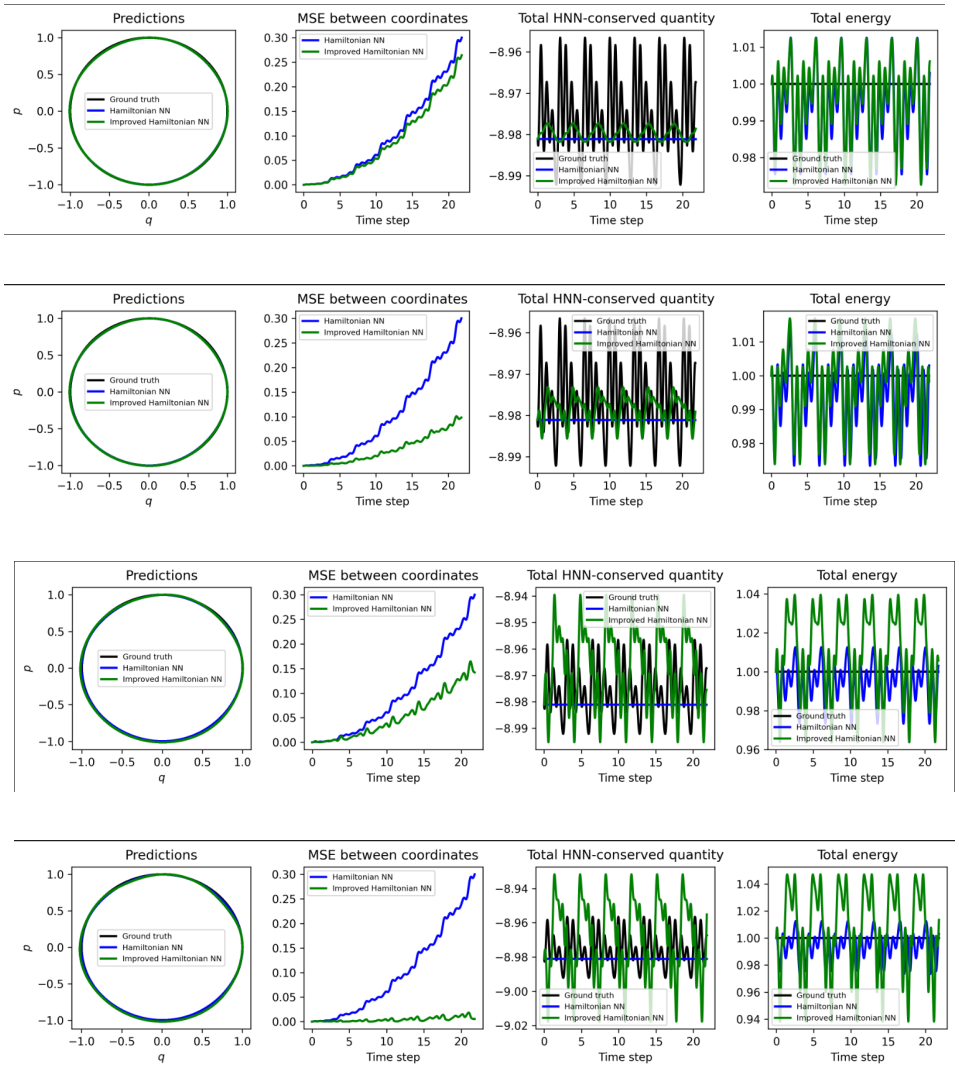


Figure: After improvement

Results with varying α

For $\alpha = [1, 10, 50, 100]$, the results for the spring problem are presented in the same order



Final comparison for spring and pendulum

With $\alpha = 100$, since it gave us the best results with respect to mean squared error in trajectory loss.

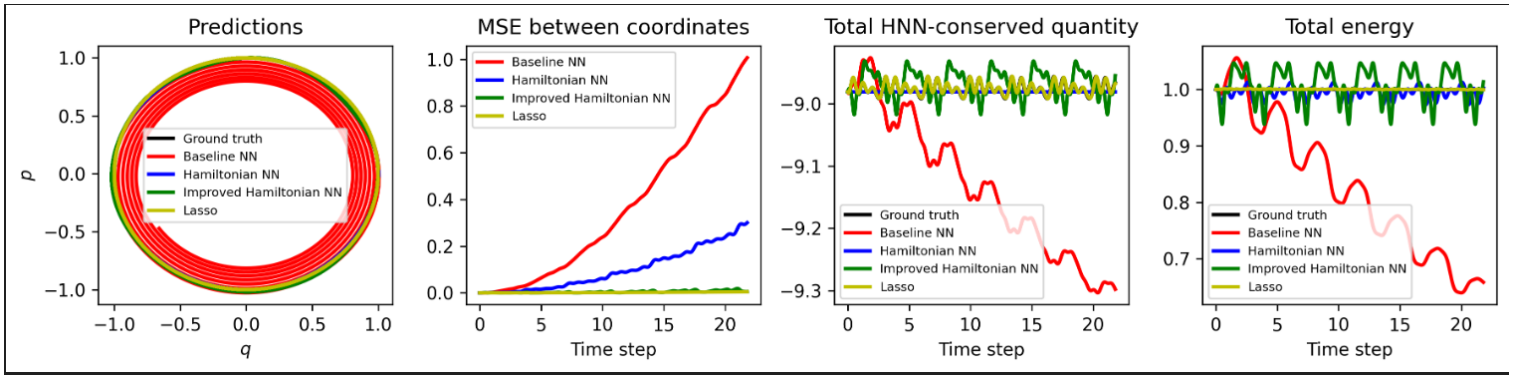


Figure: Spring

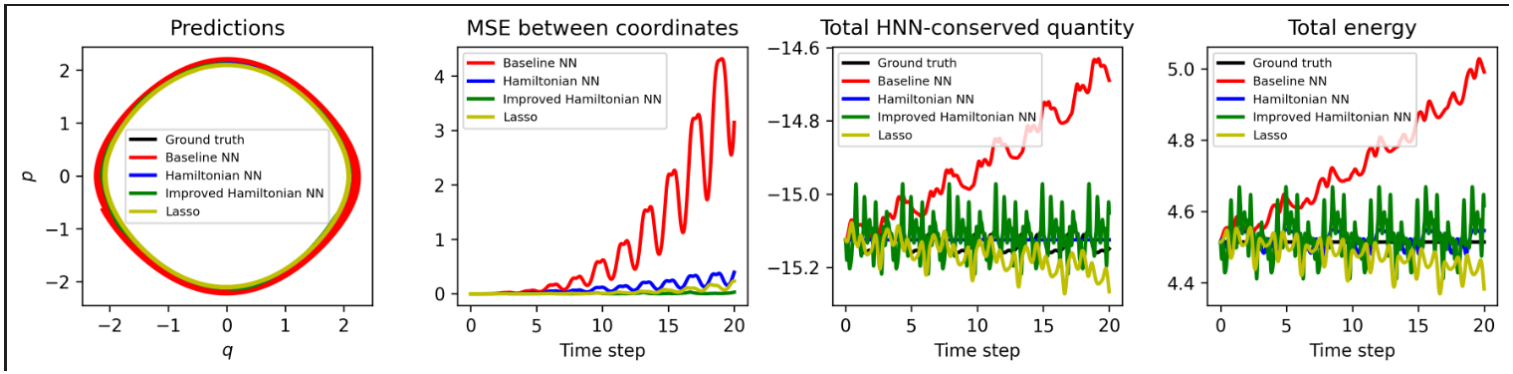


Figure: Pendulum

Damped Pendulum - Theory

For understanding the dynamics of a Real Pendulum, Lets look into how it differed from a real-pendulum

- Ideal Pendulum assumes no friction or air resistance.
- Equation of motion:

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0 \rightarrow \frac{dp}{dt} + \frac{g}{l}\sin q = 0$$

- Energy is conserved in Ideal Pendulum
- Dissipative forces like fricitonal forces acts in real-pendulum
- Energy decreases over time due to dissipation in Real Pendulum

Damped Pendulum Equation

$$\frac{d^2\theta}{dt^2} + \lambda\frac{d\theta}{dt} + \omega^2\sin\theta = 0 \rightarrow \frac{dp}{dt} + \lambda_1 p + \lambda_2 \sin q = 0$$

$$\frac{dp}{dt} = -\frac{dH}{dq} + \textit{frictional terms}$$

Damped Pendulum - Modelling

HNNs alone cannot capture the dissipative effects; So an additional fricitonal term must be incorporated. Loss function used to train HNN with friction model :

$$\mathcal{L}_{hnn+fric} = \left\| \frac{\partial H}{\partial q_t} + F_{friction}(p, q) + \dot{p}_t \right\|_2^2 + \left\| \frac{\partial H}{\partial p_t} - \dot{q}_t \right\|_2^2 \tag{2}$$

Friction is modeled by a separate neural network (Friction Network)

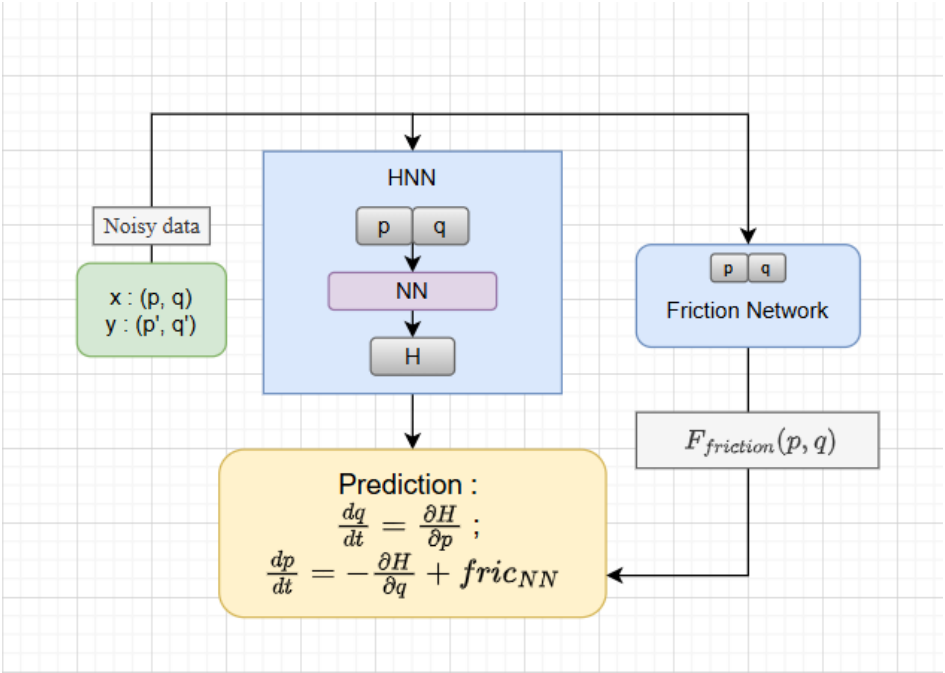


Figure: HNN with friction Network

Damped Pendulum - Results

The HNN model along with the friction network has been trained with data created using the dynamics of the real pendulum and Normal HNN has also been trained for comparison

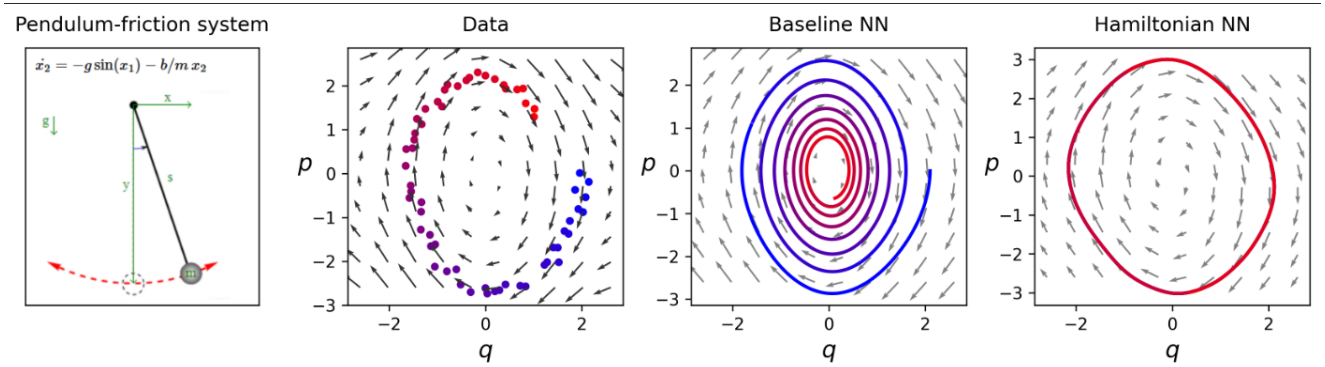


Figure: HNN-Normal vs Baseline phase-space plots for real-pendulum

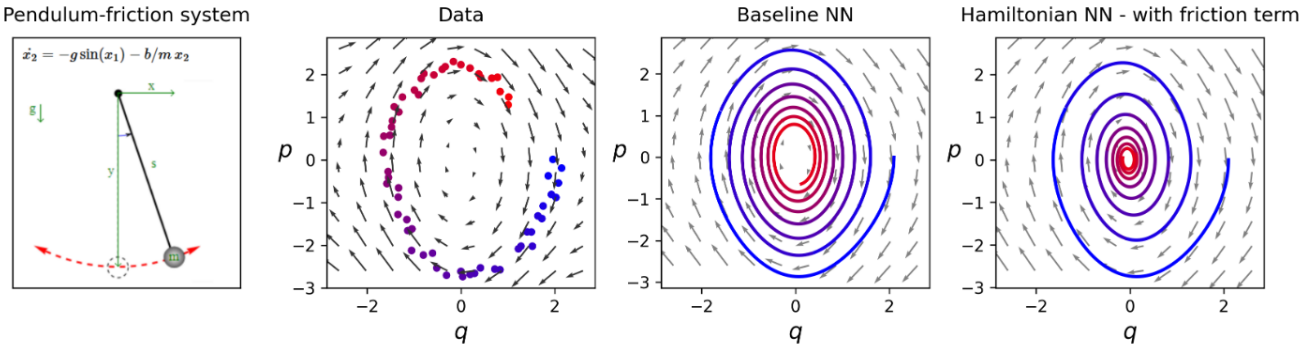


Figure: HNN-friction vs Baseline phase-space plots for real-pendulum

Damped Pendulum - Results

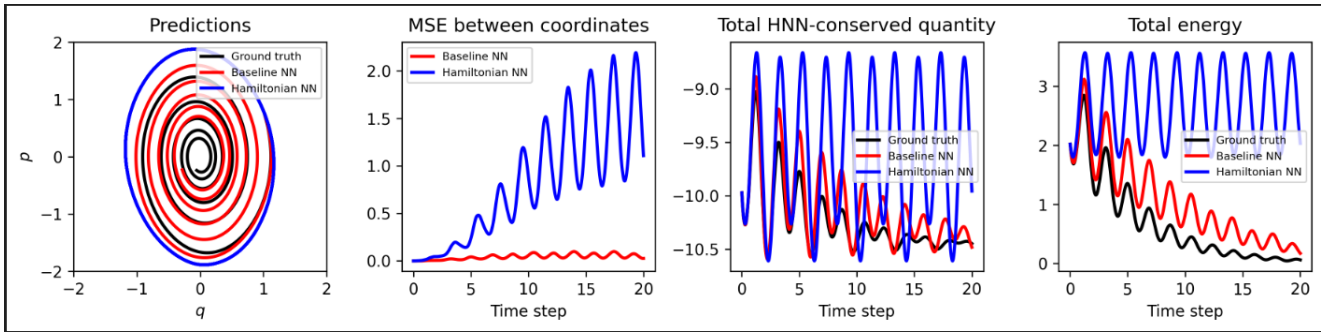


Figure: HNN-Normal vs Baseline trajectory prediction, MSE, Energy-loss

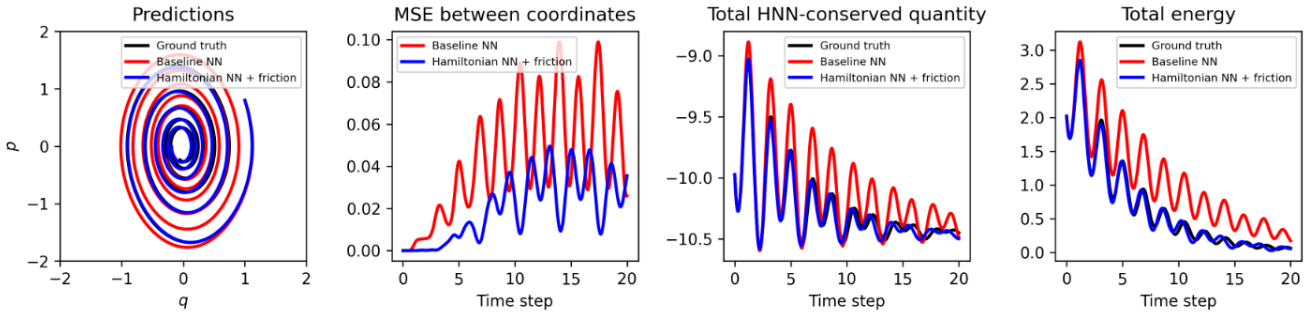


Figure: HNN-friction vs Baseline trajectory prediction, MSE, Energy-loss

HNN + friction is able to outperform both Baseline and normal HNN in predicting the decay of the damped system in both phase-space and energy to a greater extent, almost overlapping with the ground truth.

Damped Pendulum - Conclusion

Avg Energy MSE of models

Model	Energy MSE	Error
Baseline NN	41.56×10^{-3}	6.90×10^{-3}
Hamiltonian + friction NN	2.951×10^{-3}	0.22×10^{-3}
Hamiltonian NN (no friction)	1.238×10^{0}	3.27×10^{-1}

Table: Mean Energy MSE and standard deviation for different models.

- With the extra friction term, the model is able to learn the energy decay even for a long time period
- The oscillations in the energy can be due to discretisation of time-steps involved in numerical computation and can arise during a sudden change of states during decay
- Enabling HNNs with friction and other dissipative modules can help us to use HNN-based techniques in a wider class of real world and complex physical system where usually energy is not conserved

Damped Pendulum - Code snippets

Listing: Training step with friction term in PyTorch

```
# train step
dxdt_hat = model.rk4_time_derivative(x)
if not args.baseline:
    friction_dxdt_hat = torch.cat((dxdt_hat[:,0].view(-1,1), (model.
        differentiable_model.friction(x) + dxdt_hat[:,1].view(-1,1))
        ), dim=1)
else: friction_dxdt_hat = dxdt_hat
loss = L2_loss(dxdt, friction_dxdt_hat)
```

Listing: Integration step with friction step added

```
def integrate_model_friction(model, t_span, y0, **kwargs):
    def fun(t, np_x):
        dx = model.time_derivative(np_x)
        dx = torch.cat((dx[:,0].view(-1,1), dx[:,1].view(-1,1) +
            model.differentiable_model.friction(x)), dim = 1)
        return dx
    return solve_ivp(fun=fun, t_span=t_span, y0=y0, **kwargs)
```

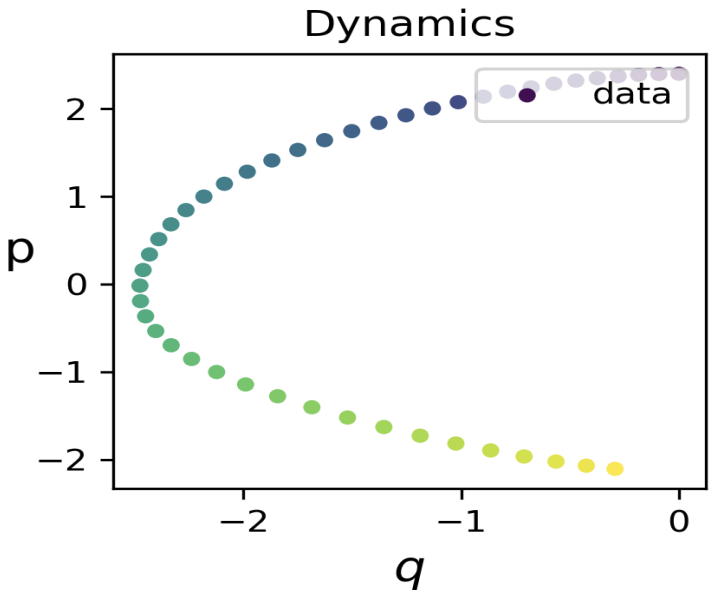
Double Pendulum System

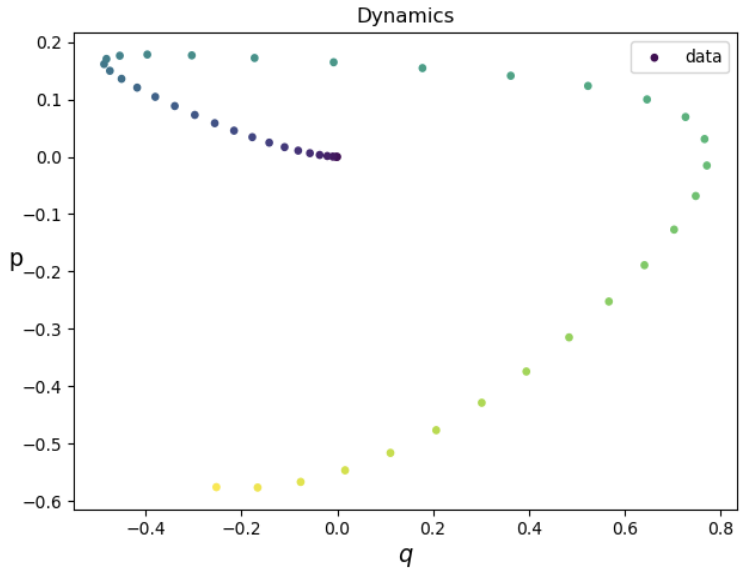
- A double pendulum consists of two point masses connected by rigid, massless rods. The first mass swings from a fixed pivot; the second mass swings from the first.
- The system exhibits extreme sensitivity to initial conditions. Even minute changes can lead to drastically different motion, making long-term prediction unreliable.

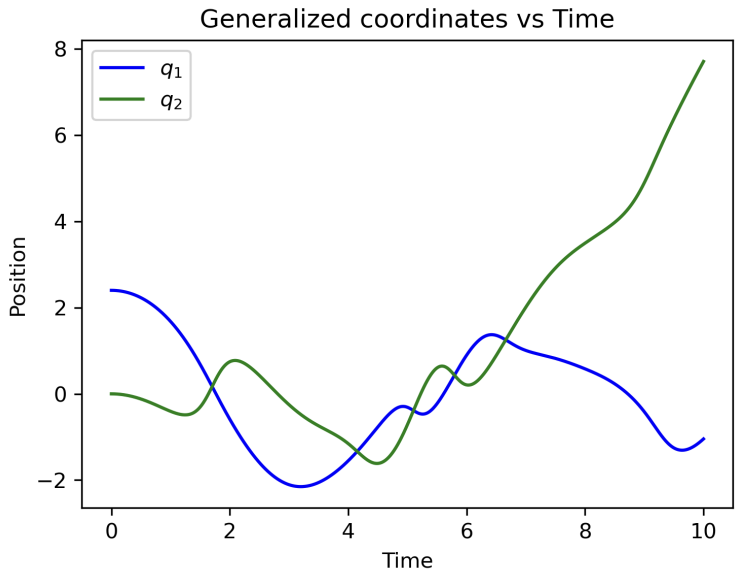
Hamiltonian

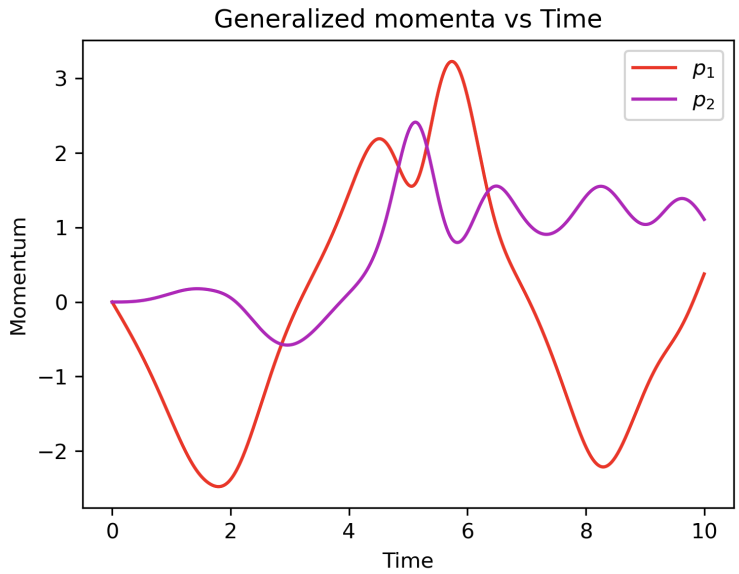
$$
\begin{aligned}
H &= T + V \\
&= \frac{1}{2} m_1 l_1^2 \dot{\theta}_1^2 + \frac{1}{2} m_2 \left(l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) \right) \\
&\quad - (m_1 + m_2) g l_1 \cos(\theta_1) - m_2 g l_2 \cos(\theta_2)
\end{aligned}
$$

Dynamics


Dynamics
data
p
q
2
1
0
−1
−2
−2
−1
0


Dynamics
data
p
q
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


Generalized coordinates vs Time
q_1
q_2
Position
Time
8
6
4
2
0
−2
0
2
4
6
8
10


Generalized momenta vs Time
p_1
p_2
Momentum
Time
3
2
1
0
−1
−2
0
2
4
6
8
10

HNN-Results

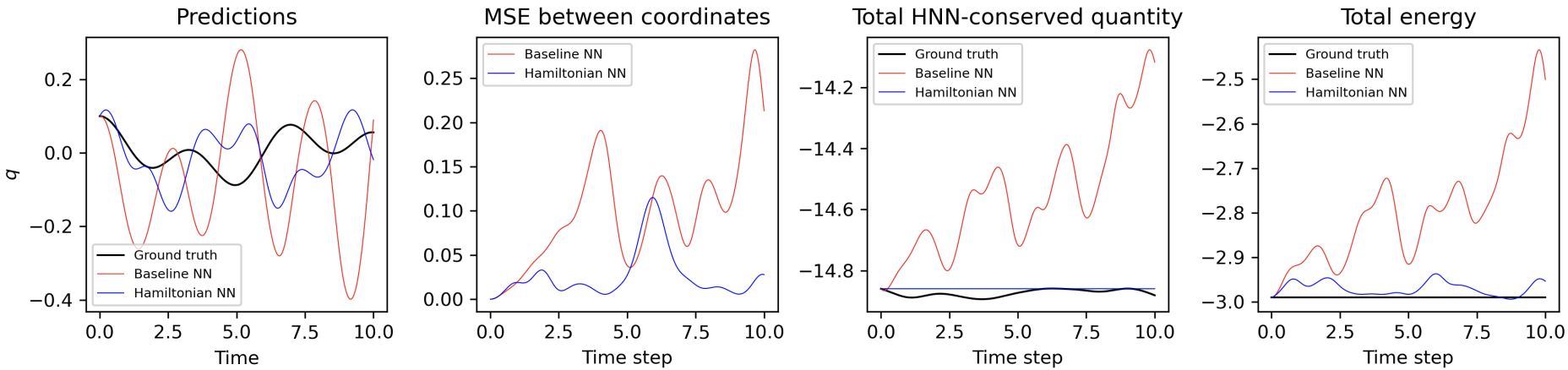


Figure: Performance of HNN over a short period

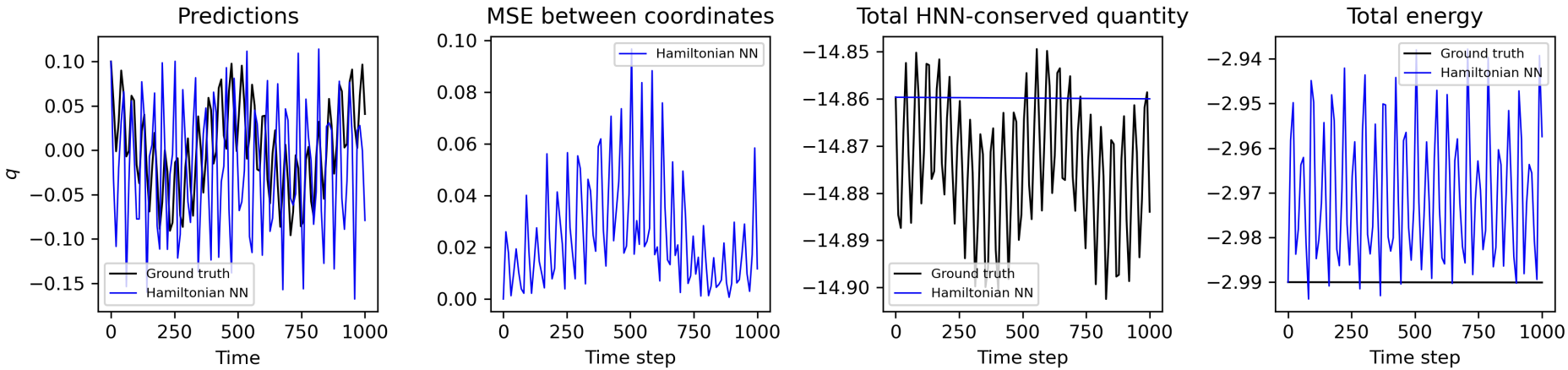


Figure: Performance of HNN over a Long period

Lagrangian of a Dynamical System

Langrangian

The Lagrangian of a dynamical system is given by:

$$\mathcal{L}(q, \dot{q}, t) = T - V$$

- T is the kinetic energy
- V is the potential energy

The system evolves according to the Euler-Lagrange equation:

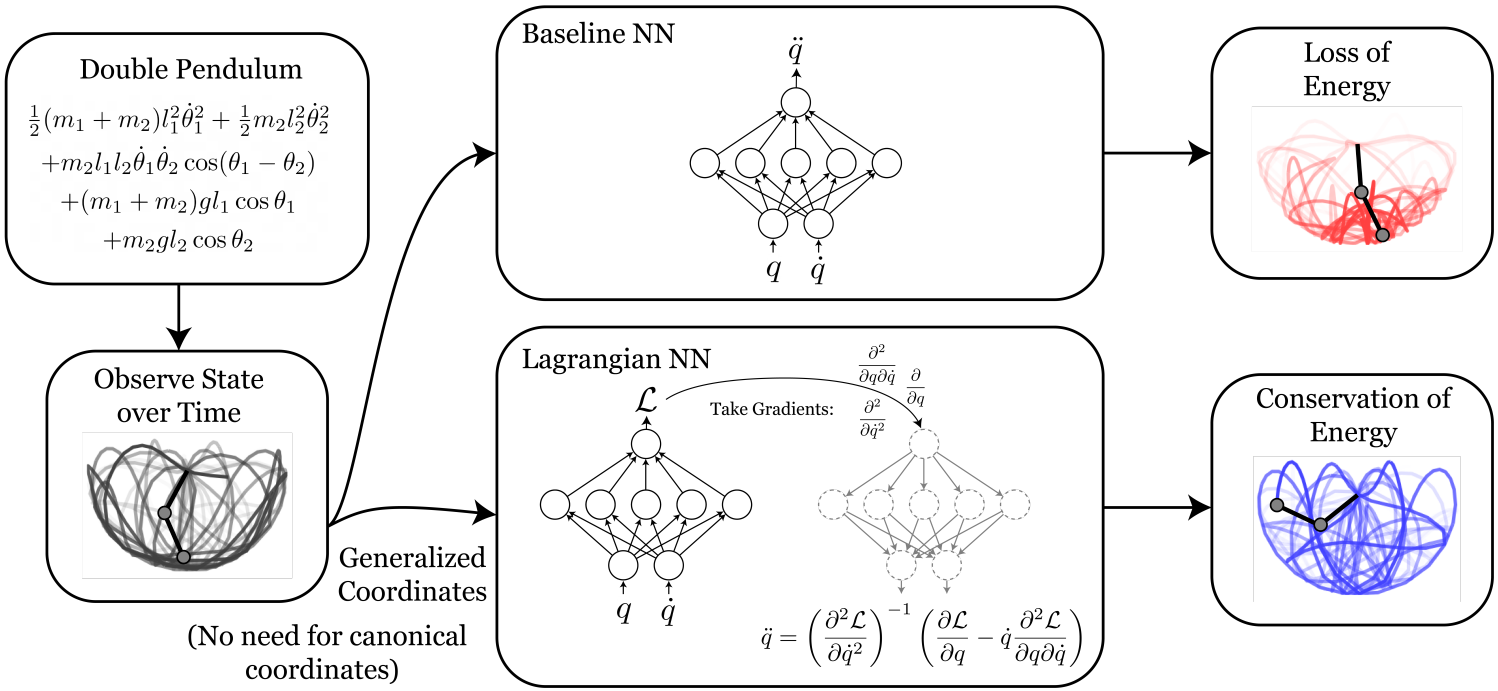
$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathcal{L}}{\partial q_i} = 0$$

Where:

- q represents the generalized coordinates
- $\dot{q}$ represents the generalized velocities

Given a state $x_t = (q_t, \dot{q}_t)$, we can compute $\ddot{q}$ and integrate to obtain the system dynamics.

Langragian Neural Network



- To train this forward model, we need to compute the inverse Hessian $(\nabla_{\dot{q}}\nabla_{\dot{q}}^{\top}\mathcal{L})^{-1}$ (using the pseudoinverse to avoid singular matrices), and then perform backpropagation.
- Obtaining the trajectory from 2nd order momentum is a very time-consuming process.

Lagrangian Neural Network Result