Hamiltonian Neural Networks

Abhay Shinde

-> Name: Abhay Shinde

-> Location: Mumbai, India

-> Branch: Information Technology

-> E-mail: theabhay.shinde@gmail.com

-> Github: https://github.com/a-b-h-a-y-s-h-i-n-d-e

-> Time Zone: UTC+05:30

1. Introduction

Traditional neural networks often struggle to learn stable dynamics due to lack of physical constraints. Hamiltonian Neural Networks (HNNs) offer a principled solution by incorporating the structure of Hamiltonian Mechanics into learning conservation laws like energy preservation.

As listed in the Deepchem model wishlist, implementing HNNs would expand the library's support for physics-informed learning. It will also provide researchers with a tool to simulate and learn from physical systems more reliably, especially in areas of molecular dynamics, materials simulation etc.

This project aims to implement the **HNN** model using Pytorch, along with an **HNNModel** wrapper that inherits from Deepchem's TorchModel class, enabling seamless integration with Deepchem's training, evaluation, and dataset handling utilities.

2. Relevant Experience and Interest

I'm an undergraduate student interested in Deep Learning and its intersection with Different domains. Implementing Hamiltonian Neural Networks within Deepchem aligns Perfectly with both my interests and current learning goals in deep learning and physics Informed modeling

Here are a few of my relevant projects:

- PainBrushAI -> Neural alchemy for generating styles
- GeoRoadExtract -> Extracting pathways from satellite images
- PongENV -> Custom environment on Reinforcement learning

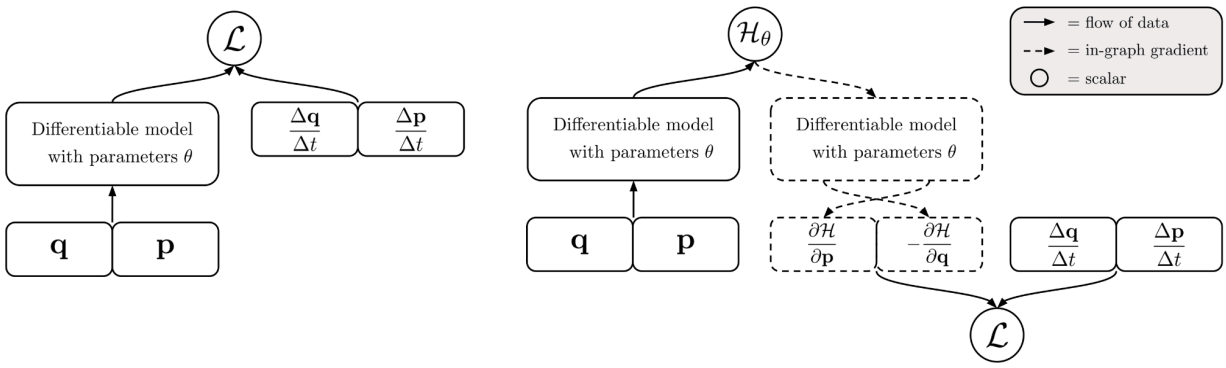
Past Contributions to deepchem

- **[** tf-test CI] [Under Review]
- [ODE Tutorial fix] [Merged]
- [Equivariance Tutorial fix] [Under Review]
- [MSA tutorial fix] [Under Review]

3. Work Plan

3.1 Introduction to HNN model

Hamiltonian Neural Networks are a class of neural networks designed to learn the Underlying dynamics of physical systems by modeling the Hamiltonian function directly. HNNs learn a scalar energy function H(q, p), where q and p represent generalized co-Ordinates (position and momentum), and derive dynamics from it using the structure Of Hamiltonian mechanics



Baseline NN

Hamiltonian NN

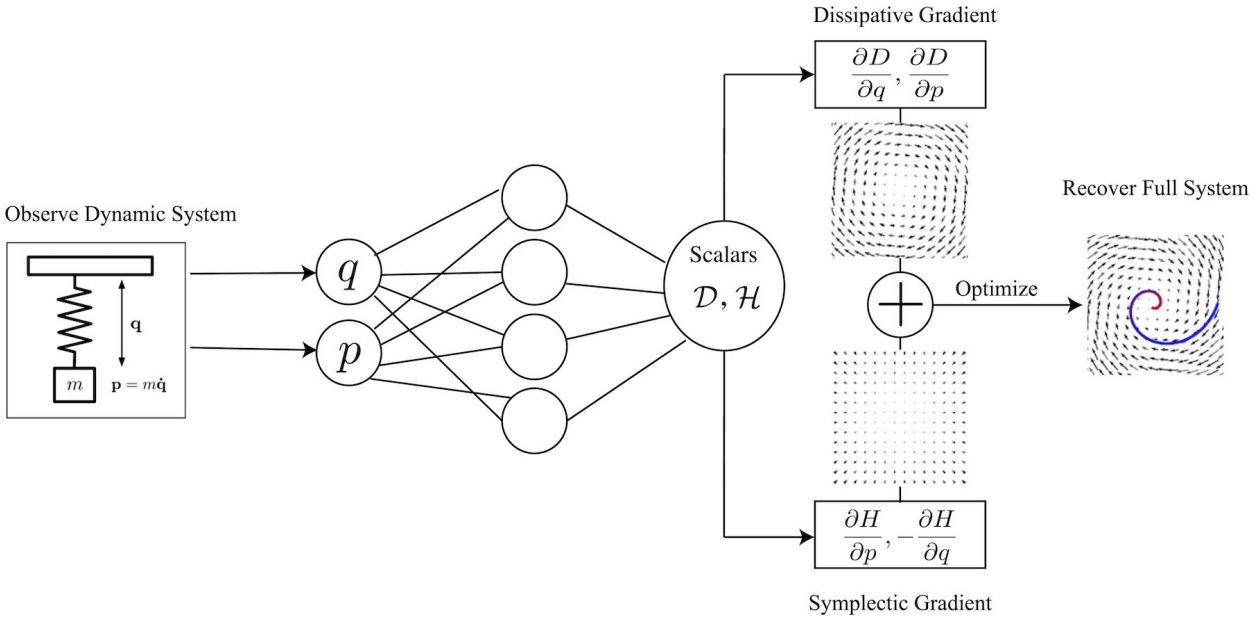
$$\frac{d\mathbf{q}}{dt} = \frac{\partial \mathcal{H}}{\partial \mathbf{p}}, \quad \frac{d\mathbf{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{q}}. \tag{2}$$

(image taken from paper)

3.2 Usage of HNN model

The HNN model is primarily used to learn and simulate the time evolution of conservative dynamical systems. Once trained on position-momentum (q, p) data, the model can be used to:

- Predict the **time derivative** (dq/dt, dp/dt) at any state in the phase space
- Generate **phase space** trajectories
- **Forecast long-term dynamics** such as spring systems, pendulums.



The time evolution is governed by:

$$\frac{d}{dt}\begin{bmatrix} q \\ p \end{bmatrix} = J\nabla H(q,p)$$

Where J is canonical symplectic matrix, ensuring that the learned dynamics are consist-
Ent with the physical laws of conservation (e.g., energy, preservation)

3.3 Design and Pseudocode

The implementation of HNN model is organized into two primary classes:

HNN class:
The HNN class is a PyTorch module that defines the core logic of the Hamiltonian Neur-
al Network. This class is responsible for learning the scalar Hamiltonian function H(q, p)

```
[ ] class HNN(nn.Module):

        def __init__(self, input_dim, hidden_dim=200):
            super(HNN, self).__init__()
            self.hamiltonian_net = nn.Sequential(
                nn.Linear(input_dim, hidden_dim),
                nn.Tanh(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.Tanh(),
                nn.Linear(hidden_dim, 1)  # the H(q, p) will return a scalar value
            )

            n = input_dim // 2 # because half are q and half are p, so we make pairs
            self.J = torch.zeros(input_dim, input_dim) # create a matrix consider as symplectic matrix
            self.J[:n, n:] = torch.eye(n)      # useful for dq/dt
            self.J[n:, :n] = -torch.eye(n)     # userful for dp/dt


        def forward(self, x):
            """
            as per the paper, values are like this [q1,.....,qn] [p1,.......,pn]
            """
            x.requires_grad_(True)

            H = self.hamiltonian_net(x)

            dH = torch.autograd.grad(H.sum(), x, create_graph=True)[0]

            return dH @ self.J.T

        # this is just optional method, which can be use to return the current H function value at given time
        def compute_hamiltonian(self, x):
            with torch.no_grad():
                return self.hamiltonian_net(x)
```

HNNModel class:
The HNNModel class acts as a DeepChem-compatible wrapper around the Pytorch-
Based HNN class. It inherits from DeepChem's TorchModel and handles training, eva-
Luation, and integration with Deepchem's NumpyDataset.

By encapsulation the lower-level Pytorch logic, the wrapper allows years to leverage
DeepChem's standardized training loops, callback systems, and dataset handling, mak-
ing the model more accessible and easier to use.

```python
class HNNModel(dc.models.TorchModel):
    def __init__(self, input_dim=2, hidden_dim=200, learning_rate=1e-3, nb_epoch=2000):
        model = HNN(input_dim, hidden_dim)
        # loss = torch.nn.MSELoss()
        loss = lambda outputs, labels, weights: torch.nn.functional.mse_loss(outputs, labels)


        super(HNNModel, self).__init__(
            model=model,
            loss=loss,
            learning_rate=learning_rate,
            output_types=["prediction"]
            # output_types=["loss"]
        )

        self.total_num_epoch = nb_epoch

    def predict_derivatives(self, dataset):
        inputs = dataset.X
        inputs = torch.tensor(inputs, dtype=torch.float32)
        return self.model(inputs).detach().numpy()

    def compute_energy_mse(self, true_energy_values, pred_energy_values):
        # this is computing between True values and predicted values
        mse = torch.nn.functional.mse_loss(torch.tensor(true_energy_values), torch.tensor(pred_energy_values))
        return mse.item()
```


[Google colab notebook link]

3.4 Testing Plan

Unit Testing:
Core components of the **HNN** class, such as:

- Output shape of the network
- Symplectic structure of the J matrix
- Correct application of hamilton's equations

HNNModel wrapper functions will be tested for:

- Compatibility with Deepchem's **NumpyDataset**
- Proper integration with the Deepchem training and evaluation pipeline
- Correct loss computation and backpropagation

These tests will be written using the **pytest** framework and included under deepchem/models/tests/ with appropriate fixtures and assertions.

3.5 Source of risk

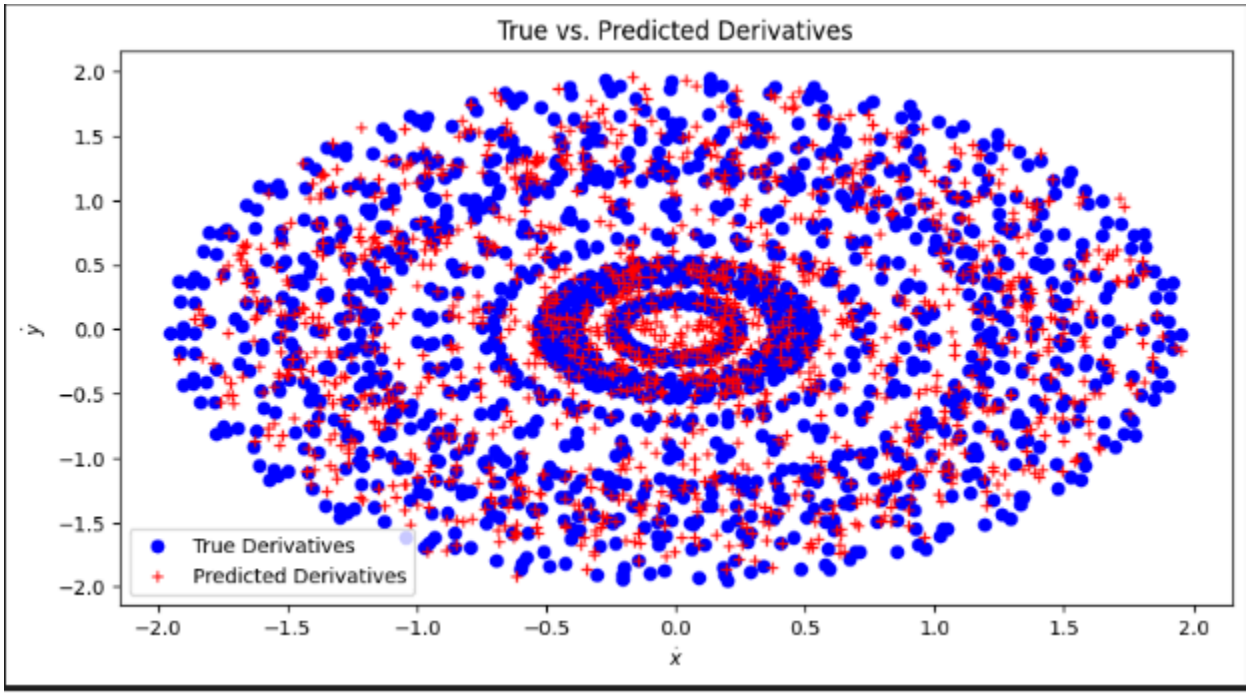
The key potential risk in this project lies in the dataset preparation and its alignment with the model's assumption. Although the original authors of the HNN paper have provided scripts [e.g dataset1, dataset2].
To mitigate this, we plan to start with the provided scripts to generate reproducible and clean datasets for initial training and testing. In addition we can also create Synthetic Datasets that simulate Hamiltonian systems with known ground truths.

3.6. Milestones

[Notebook Link]

So Currently I have implemented the Raw HNN model and trained it on the Mass-Spring dataset (link) and got some good results on the first try.

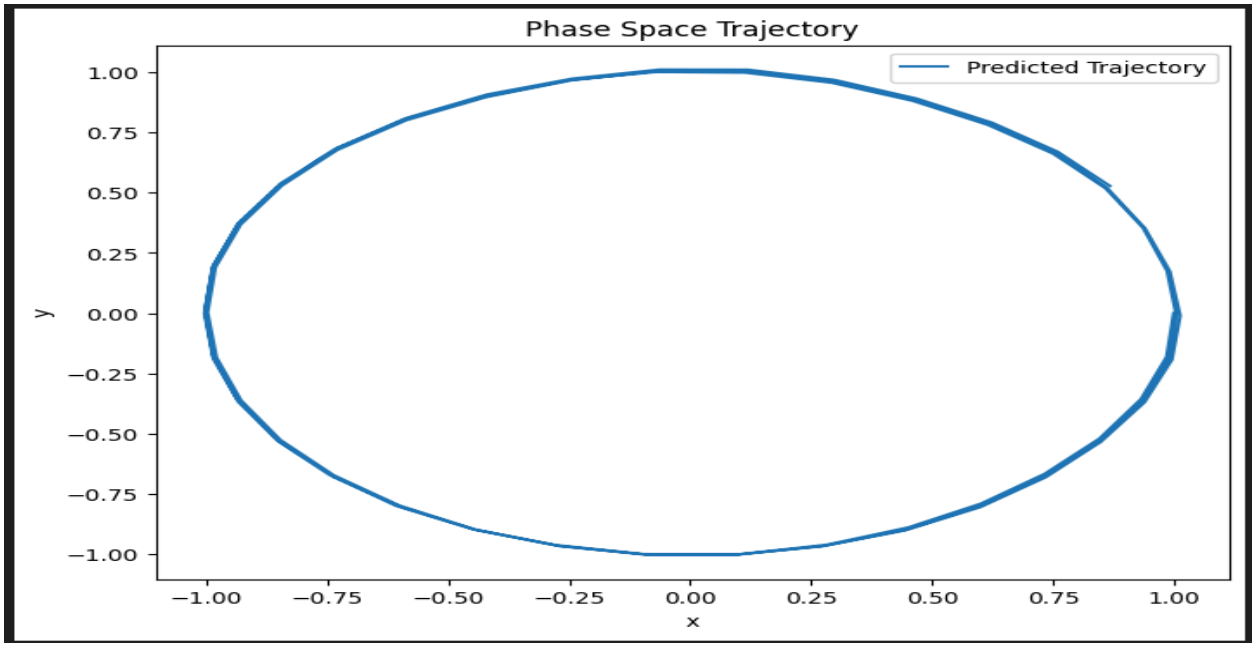
And The Results are as follows:



Trained on 2000 epochs with a learning rate as 1e-3 with 3 layers in our base Sequential model, where the hidden layer consists of 200 units (mentioned in the paper

On the first try, I got these results which were good enough to test the raw model, This

represents the True values from the dataset and predicted values from Model



This Phase Space trajectory represents how the system's state evolves over time under the Hamiltonian mechanics. Here the closed shape suggests that our model was able to preserve the energy values.

4.Timeline

Time Period	Activity
May 8 - June 1	(Community bonding period) ● Study the paper and understand the current implementation (link) ● Test some of its experiments with the raw HNN model which is currently implemented

June 2 - June 8 (week 1)	● Review the paper in detail, will understand the methods mentioned in the paper (methods refers to training the model on different experiments)
June 9 - June 15 (week 2)	● Rebuild the Raw version (current implementation), and comparing the initial results will improved model results
June 16 - June 22 (week 3)	● [PR 1] Start to build the Base HNN model and all required methods.
June 23 - June 29 (week 4)	● This week, test the Base HNN model on all experiments/examples mentioned in the paper
June 30 - July 6 (week 5)	● Keep tracking the progress of the base HNN model on experiments, what improvements can be made, the best hyperparameters for the experiments, etc. ● Also creating test cases
July 7 - July 14 (week 6)	● Testing model on deepchem NumpyDataset, (HNN takes integer input parameters in most experiments) ● Derive the plotted outputs and compare
July 14 - July 20 (week 7)	(Midterm evaluation) ● Create a wrapper around the HNN model with deepchem TorchModel support
July 21 - July 27 (week 8)	(after taking feedback from midterm evaluation) ● [PR 2] Refactor code to make it clean ● Create test cases for the HNNModel wrapper

July 28 - Aug 3 (week 9)	● Improve the test suite for both classes ● Study about the dataset (reference)
Aug 4 - Aug 10 (week 10)	● Implement raw code (like deepchem tutorial) to load the dataset and test the model. ● Plotting all the results and comparing them with previous improvements
Aug 10 - Aug 17 (week 11)	● Take feedback from mentors for the raw hnn.py code and show the results/improvements ● Conduct the final checks, refine the code structure
Aug 18 - Aug 25 (week 12)	● [PR 3] Refine the model implementation after taking feedback, Submit the final report/results ● (Optional) contribute additional resources like tested experiments, tutorials, notebooks, etc

5. Community

I have actively engaged with the Deepchem community (through discord and github discussions). These interactions have been insightful and helped me better understand the codebase and project structure

Some interactions as:

- **Aaron -** Guided me through the ODE tutorial provided clarity on the PR structure, which was especially helpful as it was my first contribution to DeepChem.
- **Rakshit -** Supported me with both the ODE and MSA tutorials, and helped me better understand DeepChem's coding and design conventions.
- **Jose -** Assisted me with the Equivariance tutorial, particularly in understanding the changes in output that occurred after my PR.
- **Shreyas** - Helped me understand the workings of DeepChem's continuous integration (CI) system, which was particularly valuable as it was my first PR involving CI-related changes.

6. Resources Required

For the development and testing of the Hamiltonian Neural Network (HNN) model, standard compute resources are sufficient. Most of the training and evaluation can be performed using freely available platforms such as Google Colab.

No paid cloud compute services are strictly required for this project. All other development and testing (integration with DeepChem, CI checks, and wrapper creation) can be performed using a local machine or CPU-backed instances.

No additional external resources or data subscriptions are needed, as synthetic datasets can be generated via scripts provided in the original HNN repository, or custom datasets can be constructed using standard physics simulation methods.

7. References

[1] https://arxiv.org/pdf/1906.01563
[2] https://greydanus.github.io/2019/05/15/hamiltonian-nns/
[3] https://github.com/greydanus/hamiltonian-nn
[4] https://www.youtube.com/watch?v=AEOcss20nDA