Restricted IOD Method using Deep Learning Function Approximation

Wasif Ul Islam
Elmore Family School of Electrical and Computer Engineering
islam25@purdue.edu
w.islam@obscuritylabs.com

December 02, 2024

1 Introduction

Problem Space:

As the greater space industry has matured over the past half-century, new markets for space based capabilities have become feasible and respectively. the demand for these capabilities have been increasing exponentially. In addition, launch costs have been decreasing at an exponential rate. It was not much long ago where a space shuttle launch would costs $1.6 billion dollars per launch. Now a constellation of small-sats can be launched at the low single-digit millions range [1]. This is leading to an exponential rise in space objects concentrating around the Earth. While the rise in space object count is not an inherently bad thing, it poses a significant need to establish capabilities surrounding *Space Traffic Management*(STM). The topic has received focused attention in the last decade, as the share of non-resident space objects (in other words, space objects we do not have defined dynamics characteristics of) have been rising drastically with respect to tagged, and cataloged resident space objects. Often times, the non-resident space objects do not have delta-v capabilities. As a result, it is imperative to develop capabilities supporting the continued effort of cataloging both known and possibly unknown space objects.

Initial Orbit Determination:

Among the greater enterprise of capabilities used in cataloging space-objects, observation based initial orbit determination (IOD) is a critical component. IOD is the process of acquiring space object orbital state given only observations and no prior knowledge of the respective space object. In general, there are two instruments that dominate the ground based observation station capability: optical telescopes, and laser ranging equipment. Optical telescopes provide measurements in Local Meridian frame, which is generally converted to geocentric right-ascension and declination as the definitive measurement. For radar ranging, range (and range rate if Doppler radar is used) along with right ascension and declination measurements are acquired [2]. With the acquired measurements, there are various methods defined, that could be used to arrive at an initial state, given the respective measurements meet method input criteria. The following is a collection of available IOD methods:

Angle-Only:

- Gauss Method
- Laplace Method

Angle and Range Measurements:

- Gibbs Method
- Herrick-Gibbs Method

The limiting factor in the above implementations of IOD are their restrictions and constraints placed on the orbit assumption. For example, the Laplace method explicitly states the assumption of a 2-body orbit, with coplanar observations [2]. These assumptions do not take into account 3rd-body affects, spherical harmonic effects of the Earth, and orbit/space craft related effects (Drag, SRP, and other lower magnitude effects). In current research, there are IOD methods that are looking to rectify model mismatch that is created as a result of these assumptions. An example is the proposed Gooding-Der Method, aimed at replacing the Lambert's solver by implementing velocity solution convergence of observations spanning multiple revolutions, using Sun universal variables [3]. Both academia and systems integrators are looking for solutions with various methods of implementation. Furfaro et al, in their contribution to the topic introduces using big data analytics, particularly machine learning over neural networks to "learn" the non-linearity of object dynamics.

2 Research Justification

As mentioned previously, classical methods of IOD provide general assumption about the object orbit. As a result, the accuracy of initial states generated by the respective methods are not at an acceptable level. This causes a heavy reliance on the robustness of orbit improvement.

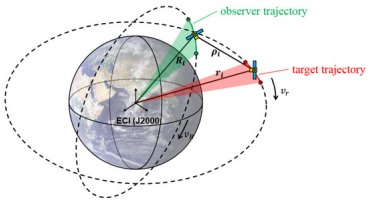


Figure 1: Display of IOD Inaccuracy [4]

As a result, there is induced demand for research being performed in closing the gap between IOD orbit accuracy before 1st orbit improvement measures are taken place.

Machine Learning Efficacy:

As density if computational power available to researchers and scientists grew, it became more feasible to study areas of approximation using data science techniques. The general term machine learning is coined from the ability for a compute device to extract patterns recognizable in its environment, whether this be from data and its respective annotations or through policy based environment interaction, also known as Reinforcement Learning. Current physics based research revolve around continuous time function approximation. Areas involving computer aided simulation, especially Computation Fluid Dynamics (CFD) and Finite Element Analysis (FEA), where large supercomputer clusters are used during product development, were the first community to invest time researching feasibility. This led to the development of Physics-Informed Neural Networks (PINN) or Theory Trained Neural Networks (TTNN), where a generalized model is trained on vast amounts of annotated data from where kernel filters identify non-linear patterns in non-linear dynamics. These physics informed neural networks have already hit commercial CAE tool ecosystem such as in Ansys,

Siemens NX, etc [5].

3 Research Context

Furfaro presented several constraints in his experiment. The author's wanted to restrict the data sample space to the simplest of orbits to allow for the model to easily approximate the specific orbit space. The following are the orbit characteristics:

- Planar GEO orbits: (0 degree inclination)
- Negligible eccentricity: [0.01 - 0.02]
- Geostationary Orbit (ECEF Right Ascension Constant)

The sample space is then further restricted by converting classical Keplerian orbital elements to Poincare orbital elements. Keplerian orbital elements are directly mapped to Poincare elements using the following equations:

$$L = \sqrt{\mu a}, \quad I = \omega + M \tag{1}$$

$$g = \sqrt{2L\left(1 - \sqrt{1 - e^2}\right)}\cos(\omega) \tag{2}$$

$$H = -g\tan(\omega) \tag{3}$$

Omitting the H term, the authors used the remaining Poincare orbital elements as the dataset label space (what the function will be predicting). On the other hand, the feature space is derived to be just the geocentric right ascension of the GEO orbit (declination is not taken into account as the inclination is 0). Note, there were no explanation by the author to what observation point is used as the paper was explaining the calculation of observer to satellite range.

The choice of label space and feature space raises several questions. Angle only IOD requires the existence of atleast 3 observations of a spacecraft to have a fully defined system. What is the analytical background behind having a single variable feature space for a 3 variable label space prediction? In addition, having a single variable feature space deems the system as under-defined. It is unknown how the author addressed system definition, as that is not discussed in the paper.

4 Author Methodology

Data Generation:

The author synthetically generates kepler elements based on the definition of the restricted orbit as mentioned in the previous section. Specifically, they draw 10000 samples from the following distributions [6]:

- Semi-Major Axis: $\mu = 42164$km, $\sigma = 100$km
- Argument of Periapsis: $\mu = 0$ deg, $\sigma = 90$ deg
- Mean Anomaly: $\mu = 0$ deg, $\sigma = 90$ deg

Note: Eccentricity was sampled from a uniform distribution spanning [0.01, 0.02]

In addition, the author mentions generation of geocentric right ascension angles, however, does not point out how to account for geographic location of the observer. During the review implementation, the geodedic coordinates of Armstrong Hall, Purdue University is used. This has significant implications on the feature and sample space as the right ascension angles have encoded the location of the observer. However, the feature space is not extensive enough to generalize that impact. Assumptions are also made to how the data was initialized as part of the training and validation loop. Review implementation will be discussed later.

Extreme Learning Machine:

The model the author selected for experimentation is the extreme learning machine (ELM) model. It is a general purpose, single layer, feed-forward network. Single layer, feed-forward networks are generally used to prototype machine learning concepts for a specific use case for its advantages involving faster compute time. The architecture involves establishing an input layer, then passing it the the single hidden layer, which is a linear combination of either a radial basis function, or a non-linear activation function with randomly generated weights, and then finally outputting the to output layer consisting of the label space. For the purpose of this experiment, the input tensor will be of size $(n, 1)$ where n is the number of samples, with the output tensor being of the size $(n, 3)$, representative of using L, I, g as the selected label space. It should be noted that the input weights and the output biases are not learnable, and should be initialized as non-learnable parameters. The following equations are representative of the sample wide operations performed in the network processing pipeline:

$$H = \begin{bmatrix} G(a_1, b_1, \mathbf{x}_1) & \cdots & G(a_L, b_L, \mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ G(a_1, b_1, \mathbf{x}_N) & \cdots & G(a_L, b_L, \mathbf{x}_N) \end{bmatrix}$$

$$H\beta = \mathbf{T}$$

Variables:

- $G(\cdot)$: Randomly generated hidden layer post passage of activation function
- X: N sized data sample
- a: Input Weights
- b: Input bias (this is set to zero for this experiment)

Only the output layer is indicated as learnable. As a result β is the only parameter that is learnable. The definition of the the optimizer is not presented as part of the paper. Assumptions are made as part of the review implementation, and general best practices are used.

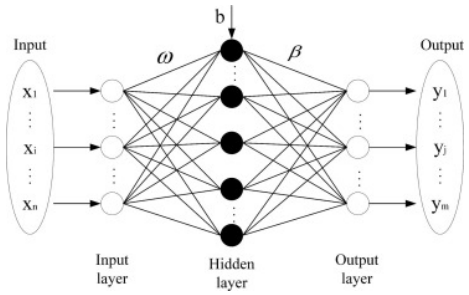


Figure 2: Graphical Representation of ELM Model [6]

Training and Validation Setup:

The author provides select hyper-parameters to indicate the initialization for the model training. The ELM model hidden layer was initialized with 18,500 neuron units, the weights and biases are randomly sampled from a standard normal distribution. The training and validation samples are split 90/10 with 9,000 samples allotted for training phase, and 1,000 samples alloted for the validation phase. An absolute loss function is used where the absolute value of the prediction is minimized by applying the gradients to the output bias. The following equation reflect the loss function:

$$\min_{\beta} \|H\hat{\beta} - T\| + \alpha\|\hat{\beta}\| \tag{4}$$

No other hyper-parameters are mentioned, such as optimizer learning rate, activation of RBF function use, number of training epochs, or batch size. All of the non-included hyper-parameters had to be assumed in the review implementation.

5 Author Results

The author generated regression charts for the three chosen Poincare elements (L, I, g) respectively. The predicted values are compared alongside with the labels. A regression line of fit is also presented with predicted and target values. The effectiveness of the function approximator is presented using correlation or R^2 value. The following figures outline the performance of the approximator as defined by the author:

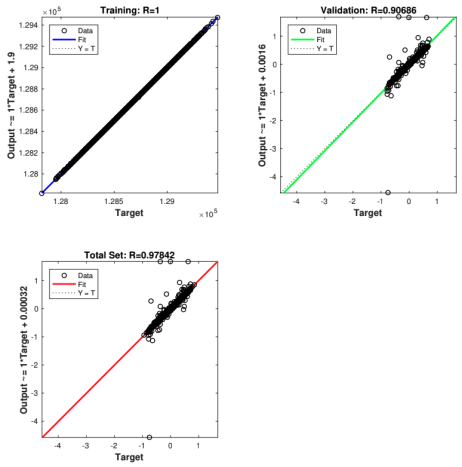


Figure 3: Regression: Poincare Element g [6]

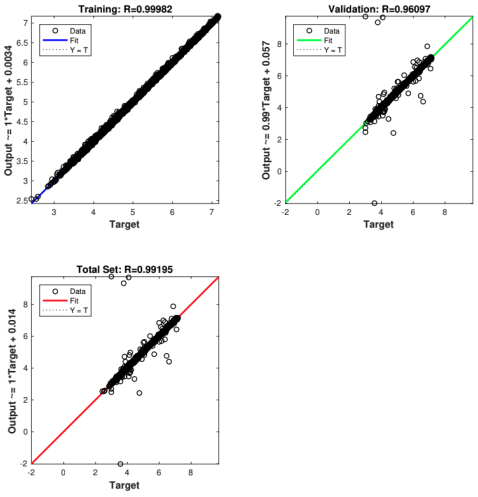


Figure 4: Regression: Poincare Element I [6]

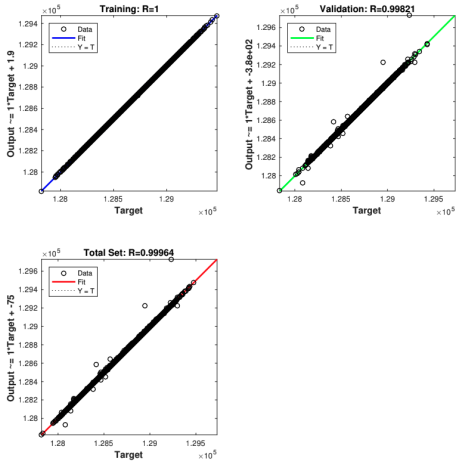


Figure 5: Regression: Poincare Element L [6]

Performance is characterized by regression correlation indicator R^2 for the Poincare element L, the training R^2 value is **0.999** and the validation R^2 value is **0.99**. For Poincare element I, the training R^2 value is **0.999** and the validation R^2 value is **0.96**. For Poincare element g, the training R^2 value is **0.91** and the validation R^2 value is **0.91**. Beyond stating the R^2 values, the author did not go into details analyzing model mutation or other benchmarks of performance.

6 Review Methodology:

Github Source: https://github.com/OnlyUseMeRocket/sda-ml

The author did not provide any source code, or guidance towards the architecture of the model or general setup of the data/observation generator. Results were generated using best effort re-implementation using Python and PyTorch learning stack. Each capability module will be presented with assumptions made at the time of implementation.

Runtime Setup:

- Operating System: Linux, Mac, Windows

- Build Dependency: UV Package Manager (https://docs.astral.sh/uv/)
- Runtime Dependency:
  - Astropy
  - IPykernel
  - Matplotlib

Note: The github repo contains a README that provides specific instruction towards re-implementing the exact experiment

Implementation:

The implementation for the experiment was split up into 3 separate modules: *data, experiment, and ML*. Each of the modules contain self containing components that is imported to a central script for modularity and ease of recreating the experiment. Any user can modify elements of the modules to suit their needs. A centralized jupyter notebook is used to integrate the separate modules together to generate the dataset, train the model, and evaluate the dataset.

Module: Data

The *data* module consists of the following self containing components:

- *dataclass.py*: Contain specific data structure definitions used by the data generator function. These data structure definitions are limited strictly to objects scoped to the data module. Instances that counts as constants are not defined here to promote modularity.

  Data Structures Defined:

  - **StateSummary**: Typed dictionary class to hold first moments for any quantity
  - **GD_LLA**: Typed dictionary class to instantiate LLA coordinates required to feed into *Astropy*
  - **Restricted_InitialOrbitStatistics**: Nested typed dictionary for representing first moments for keplerian orbital elements. Data structure instantiated to generate dataset samples

- *geo_astro.py*: Contain routines specific to orbital element calculations, and reference frame transformations from ECEF to ECI. The implementations were adapted from the $MATLAB$ implementation while adhering to *Python* data structure standards:

  Functions Implemented:

  - Convert from Geodedic Lat/Long to ITRF
  - Convert from ITRF to GCRS
  - Convert from Keplerian to Cartesian Coordinates
  - Generate Poincare Elements from Keplerian Elements

- *generator.py*: Contains singular routine for defining the dataset tensor. Returns a PyTorch dataset class required to use the dataloader interface during training and validation steps. The routine generate N samples from the defined multivariate distribution, calculates geocentric right ascension and Poincare orbital elements, export the feature and label dataset as a PyTorch *Dataset* class.

Module: Experiment

The experiment module consist of only constants necessary to initialize a discrete experiment. These constants can be changed in

the module, which would see the change reflect across the experimental setup. As of the current version of the implementation, the following constants are defined and used across modules that deem it necessary to be imported:

- Initial Orbit Parameters: 1st and 2nd moments
- Geodedic Latitude and Longitude of Armstrong Hall at Purdue
- Standard Gravitational Parameter

Module: ML

The ML module contain the implementation for the Dataset class, and the model itself. As the Dataset class is just an interface to a PyTorch tensor class, Only the neural network model will be is discussed.

The ELM model is defined as the following class definition:

```
from torch import nn
import torch

class ELM_IOD(nn.Module):
    """
    Simple extreme learning model to
    train right ascension angles
   into poincare elements
    Constructor Inputs:
        input_size: int
        hidden_layer_size: int
        output_size: int
        activation: Literal[string]
    """
    def __init__(self, input_size:
   int, hidden_layer_size: int,
   output_size: int, activation='
   relu') -> None:
        super(ELM_IOD, self).
   __init__()
        self.hidden_layer_size =
   hidden_layer_size

        # Initialize hidden layer as
    random from normal gaussian (0,
   I)
        self.input_weights = nn.
Parameter(torch.randn(input_size,
 hidden_layer_size),
requires_grad=False)
        self.output_bias = nn.
Parameter(torch.randn(
hidden_layer_size), requires_grad
=False)

        # Activation Functions
        if activation == 'relu':
            self.activation = torch.
relu
        elif activation == 'tanh':
            self.activation = torch.
tanh
        elif activation == 'sigmoid'
:
            self.activation = torch.
sigmoid
        else:
            raise ValueError("ERROR:
 Unsupported Activation Function"
)

        self.output = nn.Linear(
hidden_layer_size, output_size,
bias=True)

    def forward(self, x):
        # Hidden layer
transformation
        H = self.activation(torch.
matmul(x, self.input_weights) +
self.output_bias)
        out = self.output(H)

        return out
```

There are many different avenues of implementing the ELM model in terms of determining the output bias. I have taken the route of using an optimizer with a constant learning rate, which is trained over mini-batches of the training dataset. The output bias is also summed to the H matrix instead of integrated into the greater multi-layer perceptron for simplicity. The final output layer is then appropriately to generate the predicted label space.

Experiment Script:

The dataset is generated using the implemented label and feature generator. The ELM is initialized with 18,500 hidden nodes, with the *relu* activation function. MSE loss is used to calculate the loss between the prediction and the label. The ADAM optimizer is used to perform the *argmin* operation on the MSE loss function. The dataset is then split into training and testing datasets where 80% is used for training and 20% is used for validation. In addition, the dataset is batch seperated with each batch sized at 64 samples. This allows for the stochastic gradient initializations to avoid local minimums. The model training epoch with the lowest MSE value will be saved for validation purposes.

ELM Training Loop:

```
best_val_loss = float('inf')
best_model_state = copy.deepcopy(
    iod_elm.state_dict())
for epoch in range(1, epochs + 1):
    iod_elm.train()
    epoch_loss = 0.0
    for batch_features, batch_labels
    in train_dataloader:
        # Forward Pass
        out = iod_elm(batch_features
    )
        loss = criterion(out,
    batch_labels)

        # Backward pass and
    optimization (note we dont
    calculate gradients for weights
    and biases)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item() *
    batch_features.size(0)

    avg_loss = epoch_loss /
    train_size
    if avg_loss < best_val_loss:
        best_val_loss = avg_loss
        best_model_state = copy.
    deepcopy(iod_elm.state_dict())
        torch.save(best_model_state,
     'best_elm_state.pth')

    if epoch % 50 == 0 or epoch ==
    1:
        print(f'Epoch [{epoch}/{
    epochs}], Training Loss: {
    avg_loss:.4f}')
```

ELM Validation Loop:

```
# Instantiate Best Model State
evaluation_model = iod_elm = ELM_IOD
    (input_size=input_dimensions,
                    hidden_layer_size
    =hidden_dimension,
                    output_size=
    output_dimensions,
                    activation=
    activation_function)

evaluation_state = torch.load('
    best_elm_state.pth')
evaluation_model.load_state_dict(
    evaluation_state)

# Initialize lists to store
    predictions and labels
all_predictions = []
all_labels = []

evaluation_model.eval()
# Disable gradient calculations
with torch.no_grad():
    for batch_features, batch_labels
    in test_dataloader:
        # Forward pass: compute
    predictions
        predictions =
    evaluation_model(batch_features)

        # Append predictions and
    labels to the lists
        all_predictions.append(
    predictions)
        all_labels.append(
    batch_labels)

# Concatenate all batches into
    single tensors
all_predictions = torch.cat(
    all_predictions, dim=0)  # Shape:
     (n_test_samples, 3)
```

```
all_labels = torch.cat(all_labels,
    dim=0)              # Shape: (
    n_test_samples, 3)

# Optionally, move tensors to CPU
    and convert to NumPy for plotting
all_predictions_np = all_predictions
    .cpu().numpy()
all_labels_np = all_labels.cpu().
    numpy()
```

Regression charts are derived from predicting the 20% validation dataset against the labels associated with the input features.

7 Review Results

```
Epoch [1/1000], Training Loss: 1660769222.6560
Epoch [50/1000], Training Loss: 476976544.0000
Epoch [100/1000], Training Loss: 51914091.9360
Epoch [150/1000], Training Loss: 132729.5862
Epoch [200/1000], Training Loss: 251952.0158
Epoch [250/1000], Training Loss: 244408.6374
Epoch [300/1000], Training Loss: 180364.4895
Epoch [350/1000], Training Loss: 162245.9161
Epoch [400/1000], Training Loss: 355327.6830
Epoch [450/1000], Training Loss: 140708.7448
Epoch [500/1000], Training Loss: 58154.9292
Epoch [550/1000], Training Loss: 65445.9748
Epoch [600/1000], Training Loss: 112554.3238
Epoch [650/1000], Training Loss: 58961.3015
Epoch [700/1000], Training Loss: 141225.6588
Epoch [750/1000], Training Loss: 211067.6297
Epoch [800/1000], Training Loss: 40157.0603
Epoch [850/1000], Training Loss: 55267.9289
Epoch [900/1000], Training Loss: 40045.4809
Epoch [950/1000], Training Loss: 101827.3895
Epoch [1000/1000], Training Loss: 50263.3183
```

Figure 6: Multi-Epoch Training Loop with Loss Function Outputs

During the model training, it was observable that the model was not performing well. The MSE loss, even considering a high learning rate, experienced very large magnitudes in the order of 10^4 to 10^5 as the optimizer reached a minimum. Changing the learning rate from 0.01 to 0.1 did not change the behavior as the large gradient steps caused constant overshoots in trying to reach the global minimum. Changing the hidden neuron size by scaling it both up and down did not produce any noteworthy changes in the MSE loss magnitudes.

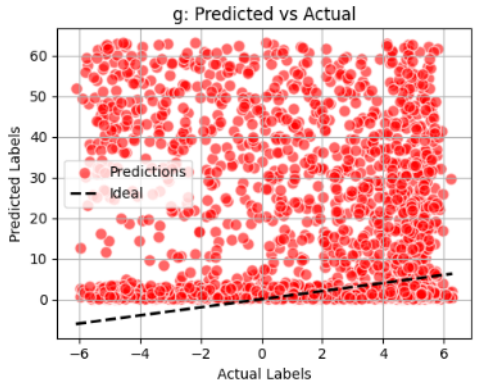


Figure 7: Regression: Poincare Element g

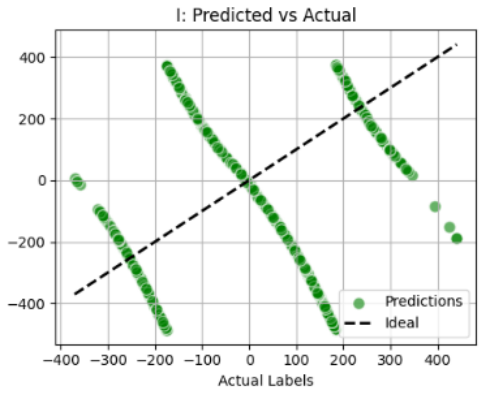


Figure 8: Regression: Poincare Element I

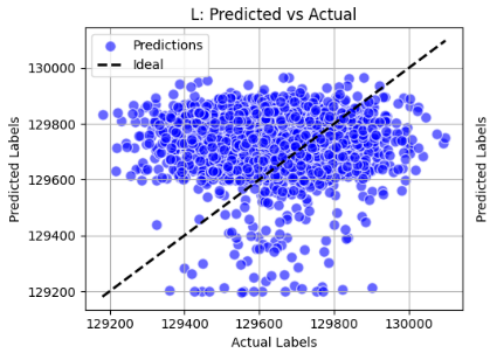


Figure 9: Regression: Poincare Element L

As observed by the high MSE loss numbers, non of the regression plot reflects the notion that the current implementation of the

ELM model is capable of performing function approximation. For any given Poincare element selected from the label, the prediction space is not describable with any pattern or trend. Additional analysis was not further performed as attempts at reducing the MSE loss values by changing hyper-parameters of the model definition did not yield improving results. Without altering the ELM model or the dataset, further analysis cannot be performed. The re-implementation concludes that model definition specifics, along with dataset definition specifics, are important to achieving relevant insights.

8 Observations

Improvements to Current Setup:

I believe the review results were drastically different than the author derived results because of the dataset, hyper-parameter, and model definitions. In terms of the dataset, I believe a feature space that involves multiple observations per label sample would provide higher probability of achieving strong performance. A similar setup to that of performing classical IOD can be emulated where 3 observations are sampled for each state definition. Having 3 observation variables to define 3 Poincare elements would provide a full-rank system from which the single hidden layer network can effectively perform kernel feature selection.

The ELM model itself can be modified to take advantage of the new defined dataset. The author analytically computes the output weight bias instead of using an optimizer to learn the bias. This has a significant overfitting drawback as the output weights are solutions to the linear equation $H\beta = T$ where T is the full dataset label space during training time. As a result, instead of the model extracting non-linear dynamics patterns, it is extracting the prior distribution pattern from which the dataset is sampled from. A multi-layer feed-forward network with learnable weights provide a more robust approach for the model to learn physical patterns existing in the generated dataset[5].

Alternate Approach:

Some alternative approach to identifying neural network efficacy in the following application would be to implement a dedicated physic informed neural network as the pretrained backbone[5]. A transformer architecture downstream of the physics informed neural network can be defined for which IOD specific datasets can be used. The dataset can also be augmented to contain relevant metadata to the right ascension observations. One examples could be multi-dimensional one-hot encoded tensor for observer location awareness. Right ascension measurements are heavily dependent on range measurements and the observer and space object position in the inertial reference frame

9 Conclusion

Increased space capability demand, development of space based economy, and sharp reduction in launch costs have created demand for innovation in developing capabilities surrounding space traffic management (STM). IOD is an important part of STM, as it is the first line of analysis after observations are made in the space object catalog workflow. Among many research areas, Furfaro presented an early stage experiment regarding non-linear dynamics function approximator to reverse map geocentric right ascension measurements to orbital elements for a restricted set of orbits, citing efficacy in other areas of research surrounding theory applied neural networks. Although the

re-implementation was not able to faithfully recreate the results presented by the author, there are multiple steps that can be taken to alter the sample space, feature space, and model architecture, that allows for the opportunity for neural networks to demonstrate their capability in aiding in IOD. Theoretical efficacy in restricted orbits can allow for label space expansion to full keplerian orbital element set or beyond. Future research into alternative model architectures, such as taking advantage of physics informed neural network (PINN) backend, and implementing a transformer model to search an expanded feature space would allow for the potential to close the gap between IOD accuracy and results from orbit improvements [5].

References

[1] R. Urban, "How much does it cost to launch a rocket?." https://spaceinsider.tech/2023/08/16/how-much-does-it-cost-to-launch-a-rocket/#elementor-toc__heading-anchor-9, 2023.

[2] D. Vallado, *Fundamentals of Astrodynamics and Applications*. Springer, 2007.

[3] F. C. Miller Collin, "Comparison of initial orbit determination methods for the use in sensor networks," in *8th European Conference on Space Debris*, 2021.

[4] E. a. Xie Hui, "A multimodal differential evolution algorithm in initial orbit determination for a space-based too short arc," *Remote Sensing, MDPI*, 2022.

[5] E. a. M. Torabi Rad, "Theory-training deep neural networks for an alloy solidification benchmark problem," *https://arxiv.org/abs/1912.09800*, 2019.

[6] E. a. Furfaro, Roberto, "Mapping sensor measurements to the resident space objects behavior energy and state parameters space via extreme learning machines," in *67th International Astronautical Congress*, 2016.

Module Code - SDA-ML

Full Code Implementation: https://github.com/OnlyUseMeRocket/sda-ml/tree/main

Data Module

dataclass.py

```
from typing import TypedDict

class StatSummary(TypedDict):
    """First moment of given parameter (Mean, Standard Deviation)"""
    Mean: float
    StandardDeviation: float

class GD_LLA(TypedDict):
    """Geodedic Latitude [deg], Longitude [deg] and Altitude (either km or m)"""
    Latitude: float
    Longitude: float
    Altitude: float

class Restricted_InitialOrbitStatistics(TypedDict):
    """Initial orbital parameter statistics for restricted orbits
    Inputs:
        SemiMajorAxis: Semi-Major Axis of Restricted Orbit [km or m]
        Eccentricity: Eccentricity of Restricted Orbit [NO DIM]
        ArgPeriapsis: Argument of Periapsis [deg]
        MeanAnomaly: Mean Anomaly [deg]

    NOTE: General reminder to always consider units when switching between m and
km
    (I'm looking at you graviatational constant !!!!!)
    """
    SemiMajorAxis: StatSummary
    Eccentricity: StatSummary
    ArgPeriapsis: StatSummary
    MeanAnomaly: StatSummary
```

generator.py

```
from data.dataclass import Restricted_InitialOrbitStatistics
import torch
import numpy as np
from experiment.constants import PURDUE_ARMSTRONG
from data.geo_astro import (
    observer_ecef_to_eci,
    observer_gd_lla_to_ecef,
```

```
    kepler_to_cartesian_restricted,
    calculate_poincare_elements
)
from ml.data import RestrictedIODDataset
from astropy.time import Time


def generate_initial_orbit_dataset_restricted(initial_orbit_stats:
Restricted_InitialOrbitStatistics, n_samples: int) -> RestrictedIODDataset:

    # Define Data Mean and Covariance
    data_mean = torch.tensor([initial_orbit_stats['SemiMajorAxis']['Mean'],
                              initial_orbit_stats['Eccentricity']['Mean'],
                              initial_orbit_stats['ArgPeriapsis']['Mean'],
                              initial_orbit_stats['MeanAnomaly']['Mean']])

    data_st_dev = torch.tensor([initial_orbit_stats['SemiMajorAxis']
['StandardDeviation'],
                              initial_orbit_stats['Eccentricity']
['StandardDeviation'],
                              initial_orbit_stats['ArgPeriapsis']
['StandardDeviation'],
                              initial_orbit_stats['MeanAnomaly']
['StandardDeviation']])

    data_variance = data_st_dev ** 2
    data_cov_matrix = torch.diag(data_variance)

    # Generate Samples
    distribution =
torch.distributions.multivariate_normal.MultivariateNormal(data_mean,
data_cov_matrix)

    # The tuple is shaped that way as the input is a torch.Size, which is a tuple
of (row, col)
    # Variables: [SMA, Eccentricity, ArgPeriapsis, MeanAnomaly]
    initial_orbit_samples = distribution.sample((n_samples, ))

    # Generate Observer Coordinates
    obs_ecef = observer_gd_lla_to_ecef(PURDUE_ARMSTRONG['Latitude'],
                                       PURDUE_ARMSTRONG['Longitude'],
                                       PURDUE_ARMSTRONG['Altitude'])

    obs_j2k = observer_ecef_to_eci(obs_ecef, Time.now())

    # Generate Object Coordinates and Poincar Elements
    obj_j2k = kepler_to_cartesian_restricted(initial_orbit_samples)
    poincare = calculate_poincare_elements(initial_orbit_samples)

    # Generate Geocentric Right Ascension
    right_ascension: list[float] = []
    geocentric_range = obj_j2k - obs_j2k
    for row in geocentric_range:
        x = row[0]
```

```
        y = row[1]
        ra = np.atan2(y, x)
        right_ascension.append(np.rad2deg(ra))

    ra_tensor = torch.tensor(right_ascension)
    ra_tensor = ra_tensor.reshape(-1, 1)
    IOD_Dataset = RestrictedIODDataset(ra_tensor, poincare)

    return IOD_Dataset
```

geo_astro.py

```
from experiment.constants import MU_EARTH_KM
import numpy as np
import torch
from astropy.coordinates import EarthLocation
from astropy.time import Time
import astropy.units as unit

def calculate_poincare_elements(orbit_samples: torch.Tensor) -> torch.Tensor:
    poincare_tensor = torch.Tensor()
    for row in orbit_samples:
        sma = row[0]
        ecc = row[1]
        argp = np.deg2rad(row[2])
        ta = np.deg2rad(row[3])

        # Poincare Elements
        L = np.sqrt(MU_EARTH_KM * sma)
        I = np.rad2deg(argp + ta)
        g = np.sqrt(2 * L * (1 - np.sqrt(1 - (ecc ** 2)))) * np.cos(argp)

        # Append to Larger Tensor
        tensor_row = torch.Tensor([L, I, g])
        poincare_tensor = torch.cat((poincare_tensor, tensor_row.unsqueeze(0)),
dim=0)

    return poincare_tensor


def observer_gd_lla_to_ecef(GD_LAT: float, GD_LONG: float, alt: float) ->
EarthLocation:
    """Calculate topocentric position vector of observer in ECEF frame
calculations done
        in meters

        Inputs:
            GD_LAT: float - Geodedic Latitude       [deg]
            GD_LONG: float - Geodedic Longitude     [deg]
            alt: float - Altitude                   [m]

        Outputs:
```

```
            R_ECEF: torch.Tensor (3,)               [km]
    """

    # Convert Angles to Radians
    GD_LAT = np.deg2rad(GD_LAT)
    GD_LONG = np.deg2rad(GD_LONG)

    distance_to_surface = 6378.137
    f = 1.0 / 298.257223563

    # Earth Eccentricity
    e = np.sqrt((2 * f) - (f ** 2))

    # Geodedic Radius
    N = distance_to_surface / np.sqrt(1 - (e ** 2) * (np.sin(GD_LAT) ** 2))

    # Calculate ECEF Position
    R_ECEF = torch.Tensor([
        (N + alt) * np.cos(GD_LAT) * np.cos(GD_LONG),
        (N + alt) * np.cos(GD_LAT) * np.sin(GD_LONG),
        (N * (1 - (e ** 2)) + alt) * np.sin(GD_LAT)
    ])

    obs_itrs = EarthLocation(
        x=R_ECEF[0] * unit.km,
        y=R_ECEF[1] * unit.km,
        z=R_ECEF[2] * unit.km
    )

    return obs_itrs

def observer_ecef_to_eci(observer_location: EarthLocation, J2K_Time: Time) ->
torch.Tensor:
    obs_j2k =
observer_location.get_gcrs(obstime=J2K_Time).represent_as('cartesian')
    obs_j2k = obs_j2k.get_xyz().value
    return torch.Tensor(obs_j2k)

def kepler_to_cartesian_restricted(orbit_samples: torch.Tensor) -> torch.Tensor:
    obj_cart = torch.Tensor()
    for row in orbit_samples:
        # Initialize Vars
        inc = 0
        raan = 0
        sma = row[0]
        ecc = row[1]
        argp = np.deg2rad(row[2])
        ta = np.deg2rad(row[3])
        orbital_param = sma * (1 - (ecc ** 2))

        # Define Perifocal State
        r_perifocal = torch.Tensor([
            (orbital_param * np.cos(ta)) / (1 + (ecc * np.cos(ta))),
            (orbital_param * np.sin(ta)) / (1 + (ecc * np.cos(ta))),
```

```
            0
        ]).reshape(3,1)

        # Rotation Matrices
        r3_raan = torch.Tensor([[np.cos(-raan), np.sin(-raan), 0],
                                [-1 * np.sin(-raan), np.cos(-raan), 0],
                                [0, 0, 1]])
        r1_inc = torch.Tensor([[1, 0, 0],
                               [0, np.cos(-inc), np.sin(-inc)],
                               [0, -1 * np.sin(-inc), np.cos(-inc)]])
        r3_argp = torch.Tensor([[np.cos(-argp), np.sin(-argp), 0],
                                [-1 * np.sin(-argp), np.cos(-argp), 0],
                                [0, 0, 1]])

        r_cartesian = torch.matmul(torch.matmul(torch.matmul(r3_raan, r1_inc),
r3_argp), r_perifocal).reshape(1,3)

        obj_cart = torch.cat((obj_cart, r_cartesian.unsqueeze(0)), dim=0)

    obj_cart = obj_cart.reshape(obj_cart.shape[0], 3)
    return obj_cart
```

Experiment Module

constants.py

```
from data.dataclass import GD_LLA
from data.dataclass import Restricted_InitialOrbitStatistics

INITIAL_ORBIT_PARAMS: Restricted_InitialOrbitStatistics = {
    'SemiMajorAxis': {
        'Mean': 42164.,
        'StandardDeviation': 100.
    },
    'ArgPeriapsis': {
        'Mean': 0.,
        'StandardDeviation': 90.
    },
    'Eccentricity': {
        'Mean': 0.015,
        'StandardDeviation': 0.001
    },
    'MeanAnomaly': {
        'Mean': 0.,
        'StandardDeviation': 90.
    }
}

# Altitude in meters
PURDUE_ARMSTRONG: GD_LLA = {
        'Latitude': 40.43157,
```

```
        'Longitude': 273.085549,
        'Altitude': 192.5
}

MU_EARTH_KM = 3.986e5
MU_EARTH_M = 3.986e14
```

ML Module

data.py

```
import torch
from torch.utils.data import Dataset

class RestrictedIODDataset(Dataset):
    """Simple dataset class so PyTorch can work with the underlying interfaces
needed for 1st party functions
        Inputs:
            feature: torch.Tensor - Feature matrix (n,1)
            labels: torch.Tensor - Label matrix (n,3)
    """
    def __init__(self, features: torch.Tensor, labels: torch.Tensor) -> None:
        """"""
        super().__init__()
        if not(len(features) == len(labels)):
            raise AssertionError("Length of Labels and Features are not equal")

        self.features = features
        self.labels = labels

    def __len__(self) -> int:
        return len(self.features)

    def __getitem__(self, index: int) -> tuple[float, tuple[float, ...]]:
        feature = self.features[index]
        label = self.labels[index]
        return feature, label
```

model.py

```
from torch import nn
import torch

class ELM_IOD(nn.Module):
    """
    Simple extreme learning model to train right ascension angles into poincare
elements
    Constructor Inputs:
        input_size: int
```

```
        hidden_layer_size: int
        output_size: int
        activation: Literal[string]
    """
    def __init__(self, input_size: int, hidden_layer_size: int, output_size: int,
activation='relu') -> None:
        super(ELM_IOD, self).__init__()
        self.hidden_layer_size = hidden_layer_size

        # Initialize hidden layer as random from normal gaussian (0, I)
        self.input_weights = nn.Parameter(torch.randn(input_size,
hidden_layer_size), requires_grad=False)
        self.output_bias = nn.Parameter(torch.randn(hidden_layer_size),
requires_grad=False)

        # Activation Functions
        if activation == 'relu':
            self.activation = torch.relu
        elif activation == 'tanh':
            self.activation = torch.tanh
        elif activation == 'sigmoid':
            self.activation = torch.sigmoid
        else:
            raise ValueError("ERROR: Unsupported Activation Function")

        self.output = nn.Linear(hidden_layer_size, output_size, bias=True)

    def forward(self, x):
        # Hidden layer transformation
        H = self.activation(torch.matmul(x, self.input_weights) +
self.output_bias)
        out = self.output(H)

        return out
```

Generate Data

Generate restricted IOD data based on the following parameters:

- a: Semi-major axis
- e: Eccentricity
- ω: Argument of Periapsis
- M: Mean Anomaly *(Although in this instance, because of super low eccentricity, it could also be true anomaly)*

In [1]:
```
%reload_ext autoreload
%autoreload 2

from experiment.constants import INITIAL_ORBIT_PARAMS
from data.generator import generate_initial_orbit_dataset_restricted

number_of_samples = 10000
iod_dataset = generate_initial_orbit_dataset_restricted(INITIAL_ORBIT_PARAMS, numbe
```

Define ELM Model and Data

Input: n size scalar array

Optimizer: ADAM optimizer

Loss: Mean Squared Loss

In [2]:
```
from ml.model import ELM_IOD
from torch.utils.data import DataLoader, random_split
import torch.nn as nn
import torch.optim as optim

# Defining ELM Parameters - From Paper
input_dimensions = 1
output_dimensions = 3
hidden_dimension = 18500
activation_function = 'relu'

# Model Initialization - ELM Model
iod_elm = ELM_IOD(input_size=input_dimensions,
                  hidden_layer_size=hidden_dimension,
                  output_size=output_dimensions,
                  activation=activation_function)

# Define Loss Function
criterion = nn.MSELoss()
```

```
# Define output layer optimizer
optimizer = optim.Adam(iod_elm.output.parameters(), lr=0.01)

# Define Dataset Batch
batch_size = 64
train_size = int(0.8 * len(iod_dataset))
test_size = len(iod_dataset) - train_size

# Splitting Train and Testing Dataset
train_dataset, test_dataset = random_split(iod_dataset, [train_size, test_size])

train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Training Loop

In [3]:
```
import copy
import torch

epochs = 1000

best_val_loss = float('inf')
best_model_state = copy.deepcopy(iod_elm.state_dict())
for epoch in range(1, epochs + 1):
    iod_elm.train()
    epoch_loss = 0.0
    for batch_features, batch_labels in train_dataloader:
        # Forward Pass
        out = iod_elm(batch_features)
        loss = criterion(out, batch_labels)

        # Backward pass and optimization (note we dont calculate gradients for weig
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item() * batch_features.size(0)

    avg_loss = epoch_loss / train_size
    if avg_loss < best_val_loss:
        best_val_loss = avg_loss
        best_model_state = copy.deepcopy(iod_elm.state_dict())
        torch.save(best_model_state, 'best_elm_state.pth')

    if epoch % 50 == 0 or epoch == 1:
        print(f'Epoch [{epoch}/{epochs}], Training Loss: {avg_loss:.4f}')
```

```
Epoch [1/1000], Training Loss: 1660769222.6560
Epoch [50/1000], Training Loss: 476976544.0000
Epoch [100/1000], Training Loss: 51914091.9360
Epoch [150/1000], Training Loss: 132729.5862
Epoch [200/1000], Training Loss: 251952.0158
Epoch [250/1000], Training Loss: 244408.6374
Epoch [300/1000], Training Loss: 180364.4895
Epoch [350/1000], Training Loss: 162245.9161
Epoch [400/1000], Training Loss: 355327.6830
Epoch [450/1000], Training Loss: 140708.7448
Epoch [500/1000], Training Loss: 58154.9292
Epoch [550/1000], Training Loss: 65445.9748
Epoch [600/1000], Training Loss: 112554.3238
Epoch [650/1000], Training Loss: 58961.3015
Epoch [700/1000], Training Loss: 141225.6588
Epoch [750/1000], Training Loss: 211067.6297
Epoch [800/1000], Training Loss: 40157.0603
Epoch [850/1000], Training Loss: 55267.9289
Epoch [900/1000], Training Loss: 40045.4809
Epoch [950/1000], Training Loss: 101827.3895
Epoch [1000/1000], Training Loss: 50263.3183
```

Evaluation and Plotting

In [4]:
```python
# Instantiate Best Model State
evaluation_model = iod_elm = ELM_IOD(input_size=input_dimensions,
                  hidden_layer_size=hidden_dimension,
                  output_size=output_dimensions,
                  activation=activation_function)

evaluation_state = torch.load('best_elm_state.pth')
evaluation_model.load_state_dict(evaluation_state)

# Initialize lists to store predictions and labels
all_predictions = []
all_labels = []

evaluation_model.eval()
# Disable gradient calculations
with torch.no_grad():
    for batch_features, batch_labels in test_dataloader:
        # Forward pass: compute predictions
        predictions = evaluation_model(batch_features)

        # Append predictions and labels to the lists
        all_predictions.append(predictions)
        all_labels.append(batch_labels)

# Concatenate all batches into single tensors
all_predictions = torch.cat(all_predictions, dim=0)  # Shape: (n_test_samples, 3)
all_labels = torch.cat(all_labels, dim=0)            # Shape: (n_test_samples, 3)

# Optionally, move tensors to CPU and convert to NumPy for plotting
all_predictions_np = all_predictions.cpu().numpy()
```

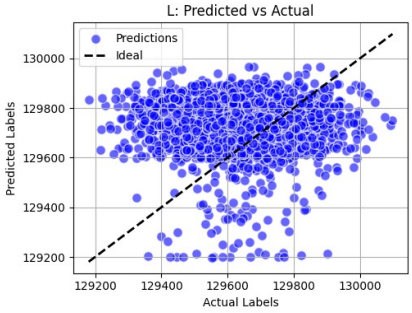
```
all_labels_np = all_labels.cpu().numpy()

print(f'All Predictions Shape: {all_predictions_np.shape}')
print(f'All Labels Shape: {all_labels_np.shape}')
```

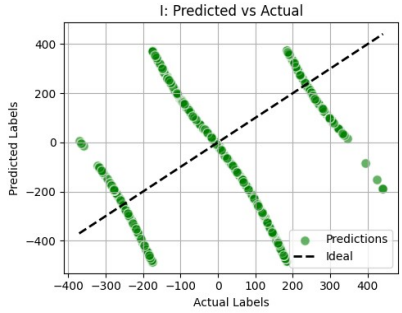
```
All Predictions Shape: (2000, 3)
All Labels Shape: (2000, 3)
```

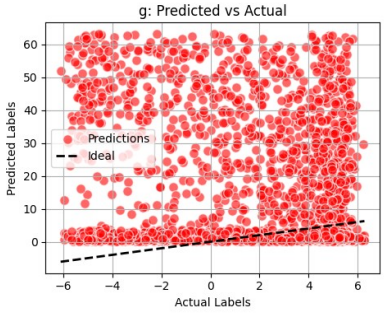
```
/var/folders/d7/d_zqr0nx3z784m45x700fnkm0000gp/T/ipykernel_79760/2300092669.py:7: Fu
tureWarning: You are using `torch.load` with `weights_only=False` (the current defau
lt value), which uses the default pickle module implicitly. It is possible to constr
uct malicious pickle data which will execute arbitrary code during unpickling (See h
ttps://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more de
tails). In a future release, the default value for `weights_only` will be flipped to
`True`. This limits the functions that could be executed during unpickling. Arbitrar
y objects will no longer be allowed to be loaded via this mode unless they are expli
citly allowlisted by the user via `torch.serialization.add_safe_globals`. We recomme
nd you start setting `weights_only=True` for any use case where you don't have full
control of the loaded file. Please open an issue on GitHub for any issues related to
this experimental feature.
  evaluation_state = torch.load('best_elm_state.pth')
```

In [6]:

```
# Plotting
import matplotlib.pyplot as plt

colors = ['b', 'g', 'r']
output_labels = ['L', 'I', 'g']

# Create subplots for each output dimension
fig, axes = plt.subplots(1, output_dimensions, figsize=(5 * output_dimensions, 4))

if output_dimensions == 1:
    axes = [axes]

for i in range(output_dimensions):
    ax = axes[i]
    ax.scatter(all_labels_np[:, i], all_predictions_np[:, i],
               alpha=0.6, edgecolors='w', s=70, color=colors[i], label='Predictions
    ax.plot([all_labels_np[:, i].min(), all_labels_np[:, i].max()],
            [all_labels_np[:, i].min(), all_labels_np[:, i].max()],
            'k--', lw=2, label='Ideal')
    ax.set_xlabel('Actual Labels')
    ax.set_ylabel('Predicted Labels')
    ax.set_title(f'{output_labels[i]}: Predicted vs Actual')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```


L: Predicted vs Actual
Predictions
Ideal
Predicted Labels
Actual Labels
130000
129800
129600
129400
129200
129200
129400
129600
129800
130000


I: Predicted vs Actual
Predictions
Ideal
Predicted Labels
Actual Labels
400
200
0
−200
−400
−400
−300
−200
−100
0
100
200
300
400


g: Predicted vs Actual
Predictions
Ideal
Predicted Labels
Actual Labels
60
50
40
30
20
10
0
−6
−4
−2
0
2
4
6