HIGH SCHOOL EDITION
Journal of Student Research
Volume 10 Issue 3 (2021)

Predicting Flying Robot Dynamics with Deep Learning

Brian Li[1] and Nathan Lambert[#]

[1]Henry M. Gunn High School, Palo Alto, CA, USA
[#]Advisor

ABSTRACT

With the rapid increase in the power of computing and technological advances in robotics, research in the field of robotics has rapidly become very expansive. Being able to accurately predict movements of a robot is vital to many applications within this field, allowing for more precise simulation and prototyping as well as more accurate control of robotic systems. In this paper, we present an adaptable neural network that accurately predicts the movement of quadcopter robotic agents which can be expanded to encompass many more robots and applications given the requisite data, producing accurate results within a small margin for error.

Introduction

Robotics has increasingly become more and more relevant in recent years, with great advancements in robotics technology advancing the field in multiple areas, such as manufacturing. Corporations like Boston Dynamics, for example, have developed ever more complex walking robots ranging from four legged pack beasts [1] to articulated human-like walkers that can run and do parkour [2]. Flying drones have also become more advanced, with growing interest in commercial delivery and other applications driving innovation. As development of human-mimicking robots continues, robots will continue to become more complicated with many moving parts.

In this field, there is one concept that is very important when designing a robot, robot dynamics. Concerned with the relationship of the robot and the forces placed upon it, it is a direct application of soft and rigid body dynamics (the motion of objects in relation to the physical factors that affect them, like force, momentum, etc.) [3] to robots. Getting a good understanding of how the application of external forces will affect a robot is important in intelligent and accurate robot design. However, it is hard to specifically predict different ways a robot will move/react given a set of input parameters without running an experiment, which researchers/engineers will be reluctant to perform if it may cause strain/damage to their valuable prototypes [4].

To help solve this issue, we present a neural network model to accurately predict the actions a robot's body will take when various forces are applied, given a set of movement data. It can extrapolate the movements a robot will take within an acceptable degree of error of up to 1%. By providing our network with more conventional movement data, it can successfully predict how a robot will react when performing much more strenuous maneuvers, like a rapid turn on a quadcopter. This would permit for more accurate simulation and control of robots compared to conventional computer simulations, making it easier to prototype new robots and improving their capacity to make decisions.

Methodology

A. Summary

To solve this problem, we need to create a predictive algorithm using robot movement data. In order to be able to accurately train the neural network, we first need to process the data to clean it. The data was preprocessed and standardized, before being placed into a custom dataset in order to allow the neural network to more accurately read

it, due to the unusual shape of the data. The neural network itself is highly adaptable, and is structured to allow complete customization over layer counts, number of inputs, and the like. In this fashion the network itself can be changed to suit different scenarios.

B. Data Structure

We utilized four different sets of movement data provided by Nathan Lambert. The first is a dataset generated by the Ionocraft, a flying microrobot (Fig 1) [5]. The remaining three datasets are generated based on a simulated version of the ionocraft using identical parameters [6]. Each dataset is 2000 elements long, consisting of 40 seconds worth of movement data, with movements in all three axes in various flight paths.

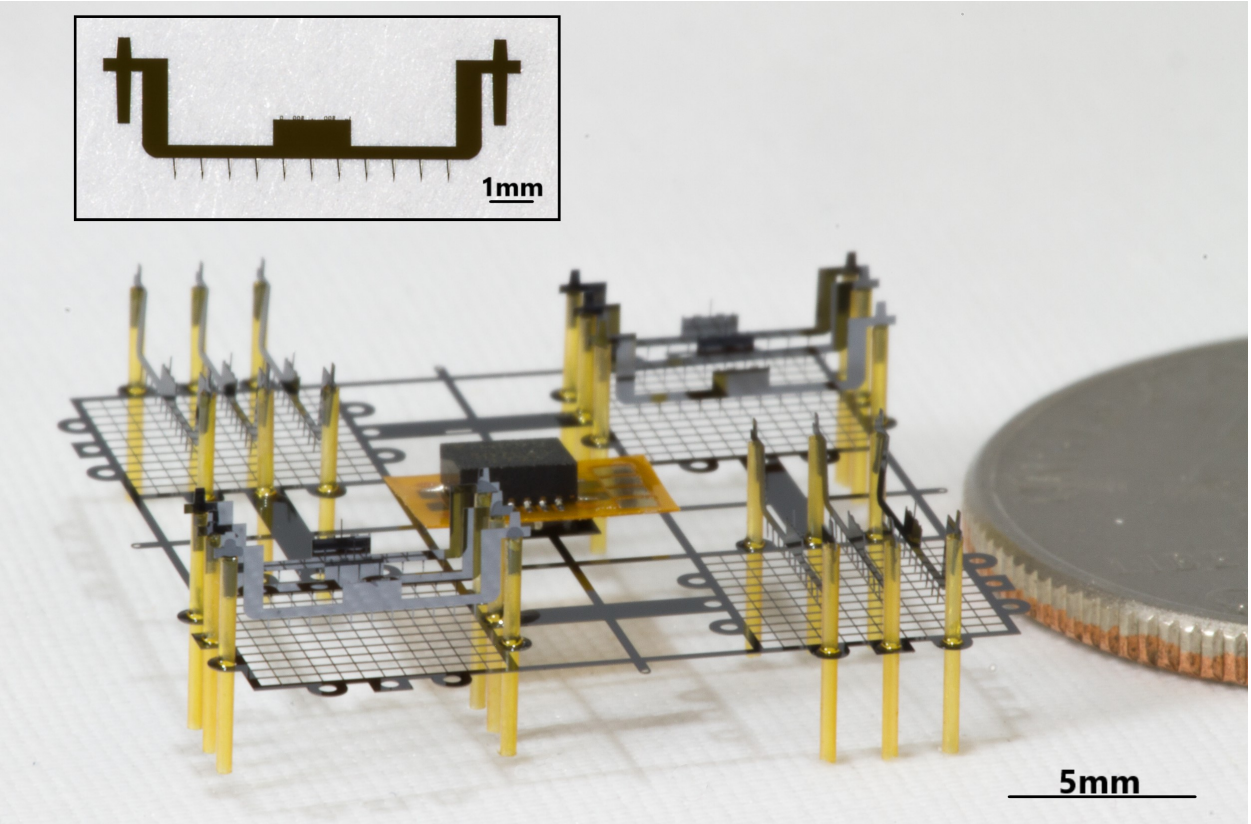

Figure 1. The Ionocraft, the robot that produced the data.

The data is fed into the neural net as a set of 10 values, corresponding to different aspects of the quadcopter's movement. The complete set of 10 corresponds to: yaw, pitch, roll, angular acceleration on X, Y, and Z axes, and the 4 motor PWM (pulse-width modulation) values. As an output, the neural network returns a set of 6 values, corresponding to the first 6 values in the input. During development, we applied two different forms of outputs and targets for testing. The first is "true-state prediction", or state-based prediction, where the data corresponds to the resulting position and orientation of the quadcopter after 0.02 seconds (Equation 1). So, in a scenario where a quadcopter moves 5 units from an original x value of 4, the output would be 9. The second is "delta-state prediction", or delta-based prediction, which corresponds to the difference/distance the quadcopter moved during the 0.02 second time frame (Equation 2). In the aforementioned scenario, the output would instead be 5, since the quadcopter moved

5 units in total. These values can then be utilized to provide an accurate representation of where the quadcopter is at any given moment.

Equation 1. The "True-state" prediction formula.

$$s_{t+1} = s_t + f_\theta(s_t, a_t)$$

Equation 2. The "Delta-state" prediction formula.

$$s_{t+1} = f_\theta(s_t, a_t)$$

$$z = \frac{x_i - \mu_i}{\sigma_i}, (\mu_i = \frac{1}{N} * \sum_{k=1}^{N} x_k, \sigma_i = \sqrt{\frac{1}{N-1} * \sum_{k=1}^{N} (x_i - \mu_i)^2})$$

Before being fed into the neural network, the data was preprocessed by first removing all rows containing non-physical outliers more than 4 standard deviations away from the mean by searching through each column. This is due to the fact that each column corresponds with a different aspect of motion, so outliers would be specific to specific columns as opposed to the entire dataset in general. By doing so, the rest of the data can be normalized without causing inaccuracies in the dataset as a whole. Then, utilizing Z-score normalization, we standardized the dataset (Equation 3).

Equation 3. The Z-score standardization formula.

This allows us to ensure that the data is manageable by the neural network, improving the training environment. Data normalization also allows us to reduce instability in the dataset by reducing noise. We then split the data into test/train sets by randomly breaking the overall dataset into two non-overlapping datasets along an 80%/20% split.

C. Network Structure

To train on the data, we utilized a simple linear neural network of 10 layers, with 50 nodes per layer. A similar, but smaller, network diagram can be seen below (Fig 2). While a more complex neural network could potentially be utilized, we found that using more layers or hidden nodes did not greatly affect the accuracy in any way while also

taking more time. For the loss criterion, we utilized MSELoss, which measures the Mean Squared Error between the predicted value and target.

D. Implementation Details

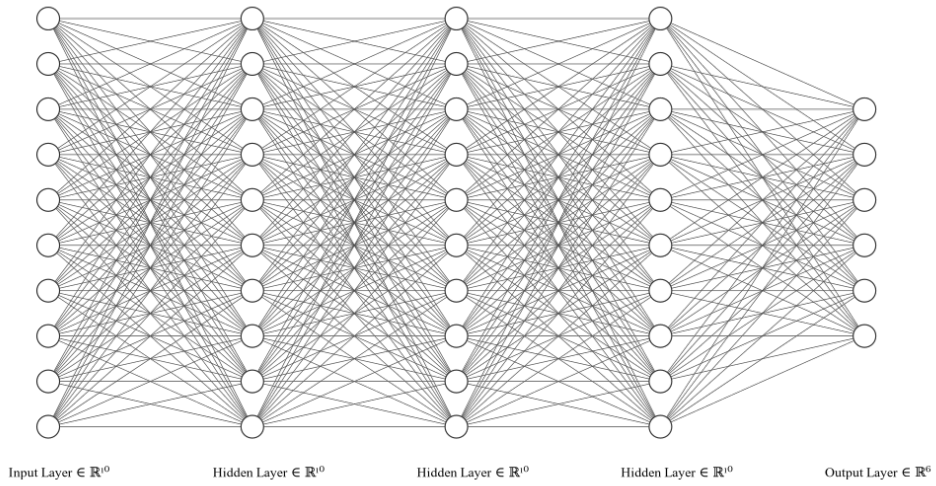

Figure 2. The structure of a neural network similar to the one used for prediction.

For training, we used a learning rate of 0.04 for our optimizer, training for 100 epochs, or iterations.
Originally, we started with a higher learning rate. However, we found that a lower learning rate provided a higher overall accuracy. We also included a learning rate scheduler that would reduce LR whenever optimization plateaued, to improve model performance.

Results / Analysis

A. Overall Results

Our results overall were extremely positive. Our results (Table 1) overall indicate a high prediction success rate utilizing state-based predictions, and positive results on the simulated datasets using delta-based prediction.

Table 1. Results of various tests conducted upon the neural network
(Percentage represents acceptable result deviation from target calculated by the equation 100*(target-predict)/target)

	Deltas 5%	Deltas 1%	States 5%	States 1%
Ionocraft Dataset	45%	9%	100%	99%
Simulated Dataset 1	100%	99%	93%	59%
Simulated Dataset 2	100%	100%	94%	61%

Simulated Dataset 3	100%	100%	92%	51%

B. Delta-based Prediction

Utilizing Delta-based prediction resulted in high accuracy rates for all three simulated datasets, giving 99-100% accuracy rates up to a 1% difference between predicted and target values (Fig 3). However, delta-based prediction was extremely inaccurate for predicting on the dataset produced by the Ionocraft, reaching only 45% accuracy with a 5% margin for error between the target and output, and 9% accuracy with a 1% margin for error.

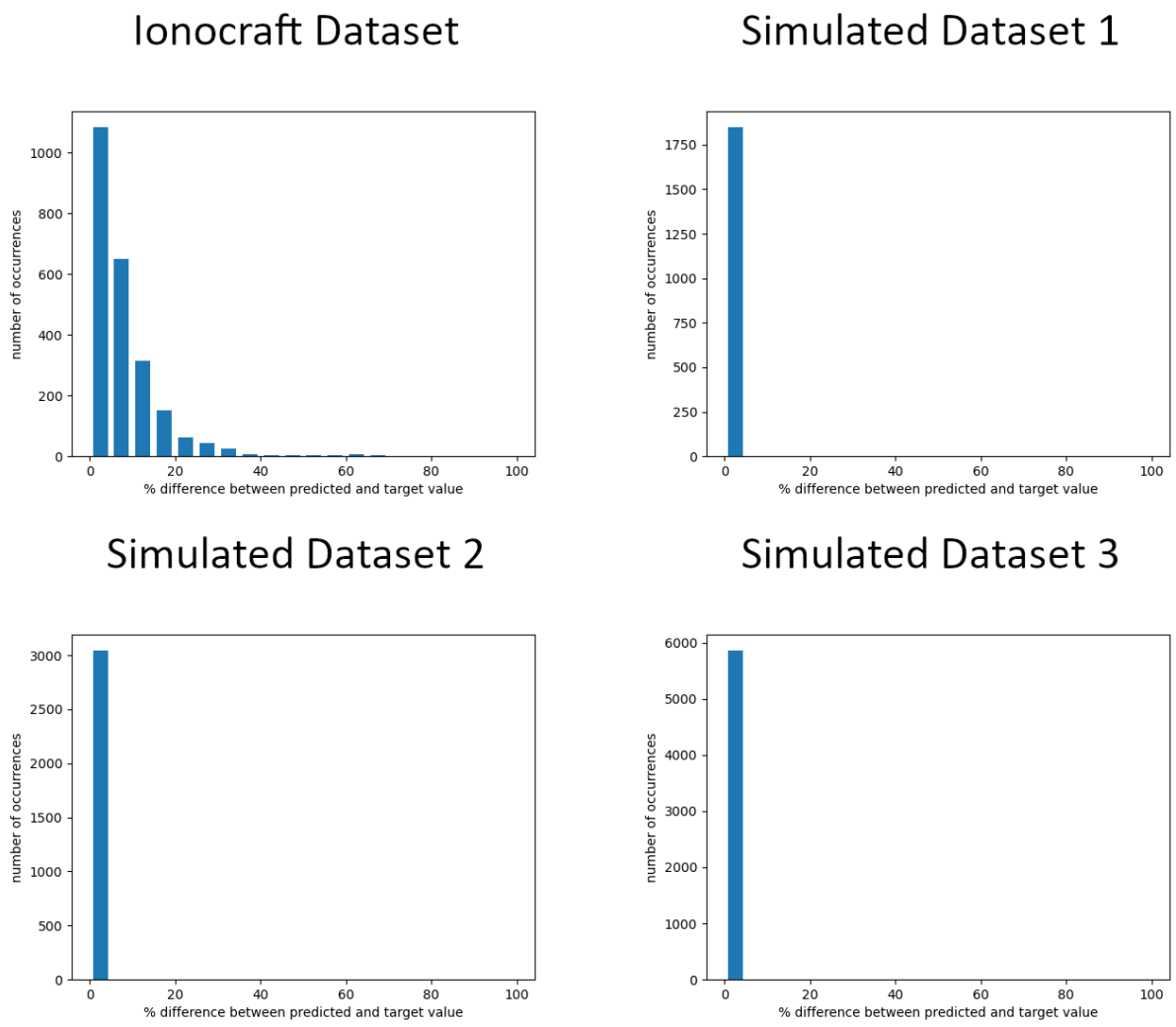

Figure 3. The % difference between results and targets using delta-based predictions.

C. State-based Prediction

State-based prediction proved to be more beneficial when applied to the real world dataset, producing around 99% accuracy with a 1% margin for error (Fig 4). It was less effective in predicting simulated robots, however, producing

results with an accuracy in the high 90%s within 5% of the target, but producing results of around 50-60% accuracy when the margin for error was shrunk.

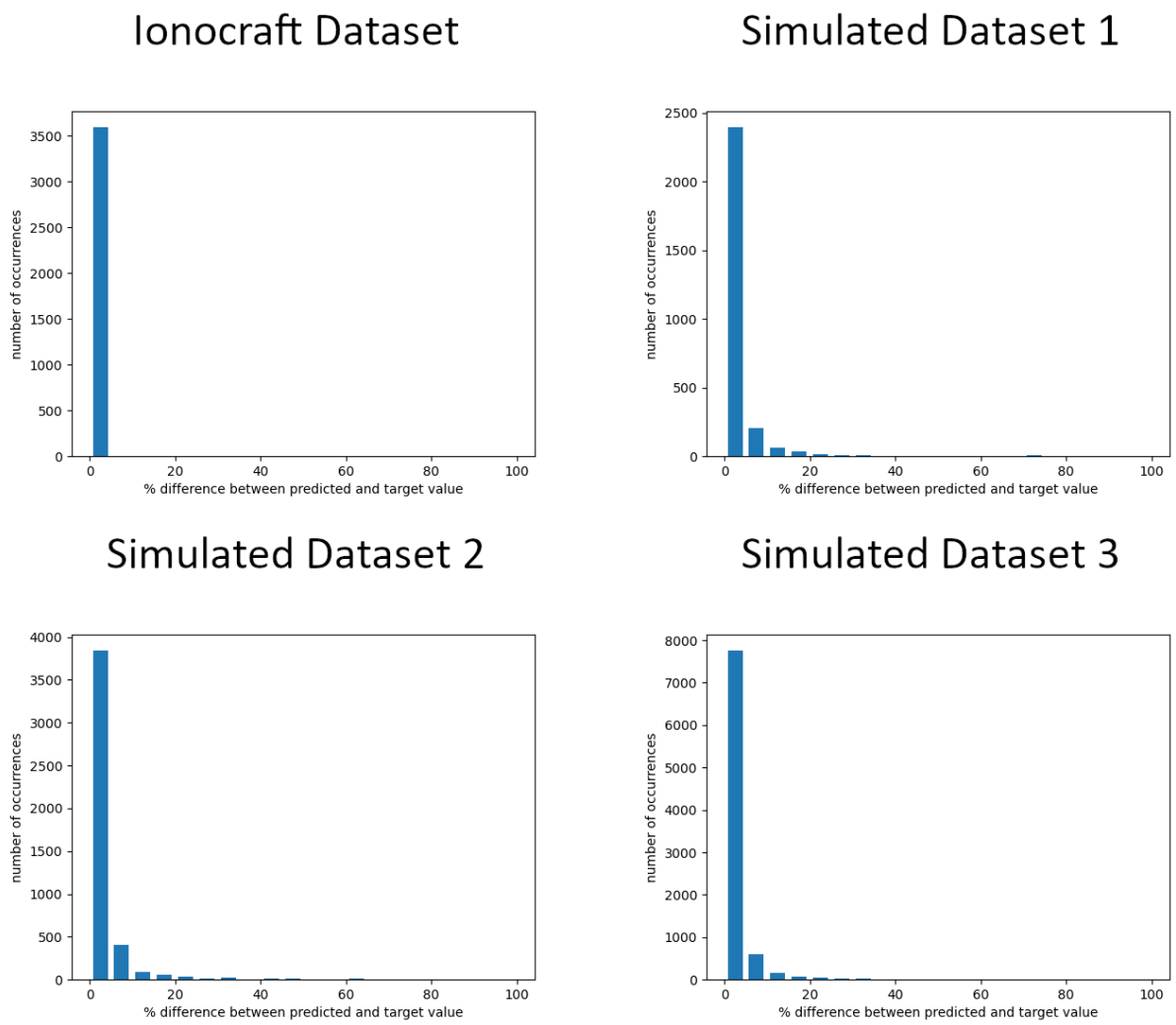

Figure 4. The % difference between results and targets using state-based predictions.

D. Analysis

As our results have shown, the two different ways of predicting dynamics work effectively on different sources of generated data, with predicting deltas working more effectively on simulated datasets and predicting states being more effective for real-world applications. We posit this is due to noise present in simulated vs real world datasets. Datasets from real robots will be more noisy than simulated datasets, meaning that predicting deltas will be more inaccurate because the immediate change in the result will be bigger. On the other hand, state-based predictions calculate an entire new position as opposed to simply the change, making the resulting error smaller. However, such a prediction method will still suffer from compounding errors, making the implementation of a Kalman Filter or more long-term trajectory predictions necessary to completely eliminate compounding errors [7]. Additionally, the use of a %-based result comparison becomes inaccurate with larger data values. Utilizing a more exact number through the use of a Mean Squared Error would allow for more precise comparisons of output and target values.

Conclusion

In our study, we produced a neural network which can successfully predict robot dynamics within a small time frame. Our neural network produced extremely positive results, reaching up to 90-100% accuracy rates on all 4 datasets with two different methods of prediction. However, these results still leave room for improvement. Because this project is incredibly dependent on the data being accurate and reliable, the inclusion of additional varied training data would increase the accuracy immensely. Additionally, the deficiencies of single-step prediction models leave our algorithm susceptible to small errors in long-term predictions. Our work would benefit greatly from the implementation of additional data normalizing functions as well as methods to reduce errors over long time periods, allowing it to more accurately predict movement in almost any circumstance.

Acknowledgments

I would like to thank Nathan Lambert for helping me with this project and providing me with the requisite data with which to train the neural network.

References

[1] *Spot | Boston Dynamics*. https://www.bostondynamics.com/spot. Accessed 8 Apr. 2021.
[2] *Atlas® | Boston Dynamics*. https://www.bostondynamics.com/atlas. Accessed 8 Apr. 2021.
[3] Smith, J.O. *Physical Audio Signal Processing,* http://ccrma.stanford.edu/~jos/pasp/, online book, 2010 edition, accessed 7 Apr. 2021.
[4] Nanyang Technological University. "Scientists develop 'mini-brains' to help robots recognize pain and to self-repair." ScienceDaily. ScienceDaily, 15 October 2020. https://www.sciencedaily.com/releases/2020/10/201015101812.htm.
[5] Drew, Daniel S., et al. "Toward Controlled Flight of the Ionocraft: A Flying Microrobot Using Electrohydrodynamic Thrust With Onboard Sensing and No Moving Parts." *IEEE Robotics and Automation Letters*, vol. 3, no. 4, Oct. 2018, pp. 2807–13. *DOI.org (Crossref)*, doi:10.1109/LRA.2018.2844461.
[6] Lambert, Nathan. *Natolambert/Dynamicslearn*. 2018. 2021. *GitHub*, https://github.com/natolambert/dynamicslearn.
[7] Lambert, Nathan O., et al. "Learning Accurate Long-Term Dynamics for Model-Based Reinforcement Learning." *ArXiv:2012.09156 [Cs]*, Dec. 2020. *arXiv.org*, http://arxiv.org/abs/2012.09156.
[8] *NN SVG*. http://alexlenail.me/NN-SVG/index.html. Accessed 8 Apr. 2021.

Appendix

The following is a part of the neural network and training loop. Please contact the author for the full algorithm.

```
import torch as torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from torch.utils.data import random_split
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from collections import OrderedDict
import math

class predNet(nn.Module):
        def __init__(self, n_in, hidden_w, depth, n_out):
        super(predNet, self).__init__()
        self.n_in = n_in
        self.hidden_w = hidden_w
        self.depth = depth
        self.activation = nn.Softmax()
        self.n_out = n_out
        layers = []
        layers.append(('dynm_input_lin', nn.Linear(self.n_in, self.hidden_w)))
        layers.append(('dynm_input_act', self.activation))
        for d in range(self.depth):
        layers.append(('dynm_lin_' + str(d), nn.Linear(self.hidden_w, self.hidden_w)))
        layers.append(('dynm_act_' + str(d), self.activation))
        layers.append(('dynm_out_lin', nn.Linear(self.hidden_w, self.n_out)))
        self.features = nn.Sequential(OrderedDict([*layers]))

        def forward(self, x):
        x = self.features(x)
        return x

model = predNet(10, 50, 2, 6)
# print(model)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', factor=0.5, patience=2)
test_errors = []
train_errors = []

# load data
trainLoader = DataLoader(train_set, batch_size=bs, shuffle=True, collate_fn=my_collate)
testLoader = DataLoader(test_set, batch_size=bs, shuffle=True, collate_fn=my_collate)

for epoch in range(epochs):
        # testing and training loss
        train_error = 0
        test_error = 0
        for i, data in enumerate(trainLoader):
        inputs, targets = data
        inputs = Variable(inputs, requires_grad=True)
        targets = Variable(targets, requires_grad=True)
```

```
optimizer.zero_grad()
predict = model(inputs)
loss = criterion(predict, targets)
train_error += loss.item() / len(trainLoader)
loss.backward()
optimizer.step()
for i, data in enumerate(testLoader):
inputs, targets = data
outputs = model(inputs)
loss = criterion(outputs.float(), targets.float())
test_error += loss.item() / (len(testLoader))
# step lr scheduler
scheduler.step(test_error)
print(f"Epoch {epoch + 1}, Train loss: {train_error}, Test loss: {test_error}")
train_errors.append(train_error)
test_errors.append(test_error)
```