Homework #3 - Machine Learning for Robotics (RBE 577) Dubin Airplane Trajectory RNN

Eric Viscione, Paul Crann
RBE577: Machine Learning For Robotics

Introduction:

Dynamical models of systems rely on knowledge of the preceding states to be able to accurately and realistically predict the next state. A 3D Dubin trajectory is a set of 3D points that following a set of constraints will generate a path from point A to point B. The Dubin model and therefore the RNN model needs the desired positon and starting position, heading and attitude to be able to create this model. Having a pre-trained neural network that is able to quickly and accurately generate these paths.

RNN Model:

RNN:

An RNN model is a type of neural network that relies on a hidden layer to retain information about the previous inputs to its system. This is ideal for a dynamical model, such as a Dubin trajectory, because it takes the output of its last state and uses that to calculate its new state. The state output of this model is the xyz coordinates for each point in the trajectory.

Data Used:

The data used was generated by using the classic deterministic algorithm. This algorithm took in the current position, heading, and desired position and outputted the trajectory as xyz points along the path. For training, there were 250,000 points used to train the model. The points consisted of a certain number of points per trajectory. This ranged from 50 to 100. When training the mode, the data was padded to bring all trajectories to the same lengths.

Regularization:

Regularization was used for this assignment. This came in the form of an L2 loss term based on the parameters weights. Adding this term allowed us to introduce a small amount of bias into our training performance in exchange for a large decrease in variance in our testing performance. In other words, this L2 term helped us avoid overfitting to our training data.

Hyperparameters:

```
Hidden Layer Size: 32
Initial Learning Rate = 0.0005
Number of Epochs = 400
L2 Regularization = 0.0001
LR scheduler step size = 10
LR scheduler decay gamma = 0.70
```

We used the above hyper parameters to fine-tune our model. They were derived through trial and error until further tweaks did not improve the system's performance. The learning rate scheduler and number of epochs had the biggest impact on our training. Relatively large learning rates resulted in flatlined performance very quickly, while relatively small learning rates resulted in slow training. The scheduler allowed us to change the learning rate every 10 epochs to 70% of the previous value. This enabled our model to take advantage of the quicker training in the beginning, while not overshooting more optimal parameter sets later. Due to the models continuous improvement with smaller learning rates, we needed to train for 400 epochs to reach an optimal solution.

Model Architecture:

We implemented a one-to-many RNN architecture for sequential data processing with a 4-dimensional input space and 3-dimensional output space. The model uses a single-layer vanilla RNN with ReLU activation, and 32 hidden nodes. The output is generated with a two layer fully connected neural net, separated by ReLU activation, transforming the hidden states to output coordinates.

Lessons Learned:

During the training of this model, we quickly figured out the importance of regularization. Without any significant regularization, the model overfit and was untrainable after 10,000 points. With regularization we were able to train it with upwards of 250,000 points and created a much more accurate model.
Additionally, we learned the benefits of using a properly tuned learning rate scheduler. This scheduler allowed us to bring our testing loss down from ~2000 to ~100 within a reasonable training window.

Results:

As observed in the TensorBoard images below, our model nicely converges with our training and testing losses being minimized throughout training. As observed in figure 3, our model generally predicts the correct trajectory. There are some variations in runs 3 and 7, where we output unintended deviations from the optimal path, but overall the trajectories look reasonable.

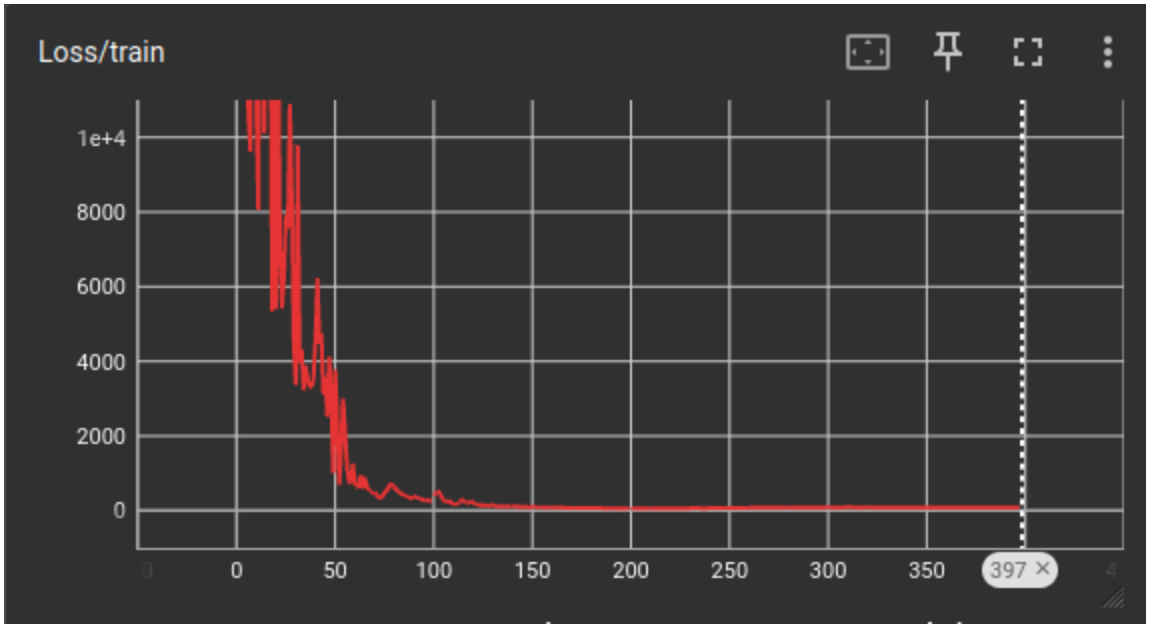

Figure 1: Training Loss vs Epoch

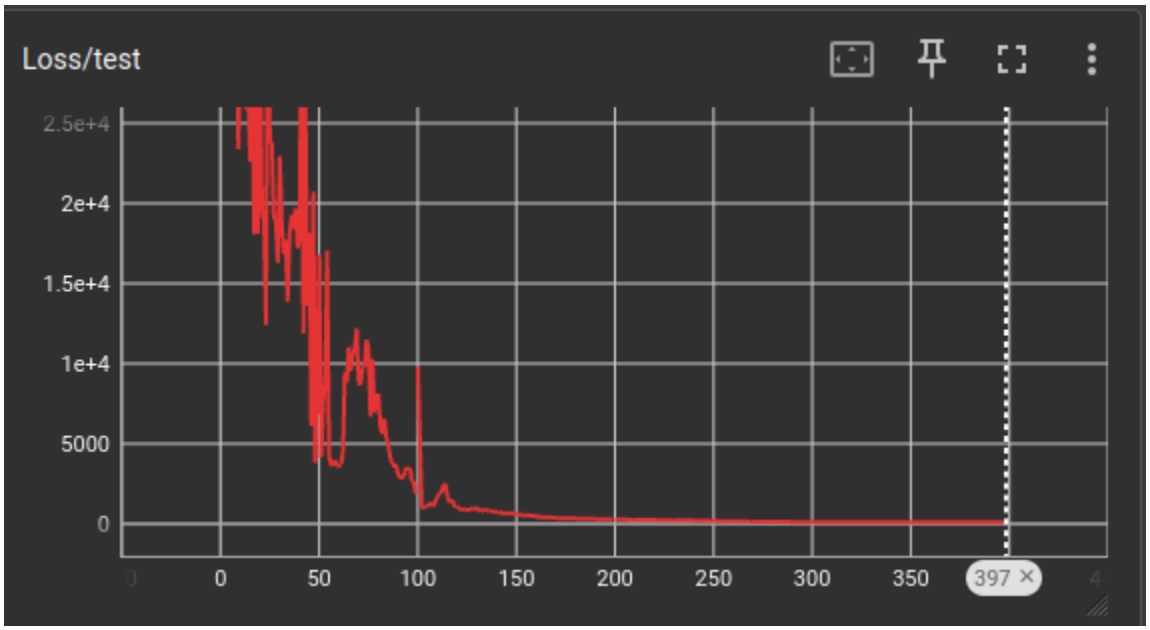

Figure 2: Testing Loss vs Epoch

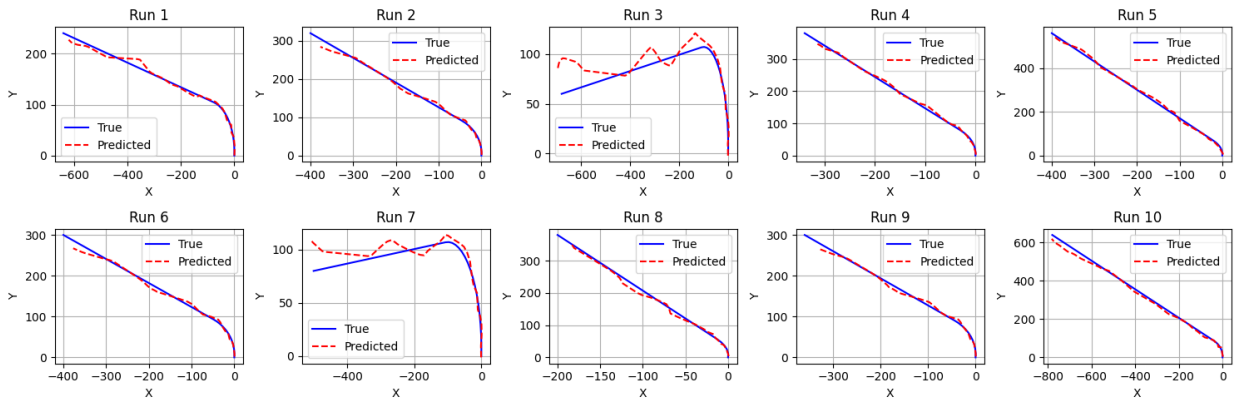

Figure 3: Testing examples