RBE 577: Homework 3 Report

Azzam Shaikh
Oct. 27, 2024

Introduction

The purpose of this assignment is to train a recurrent neural network to predict the trajectory of a Dubin's airplane. A Dubin's airplane is an airplane that follows a Dubin's path, i.e. the shortest curve connecting two points in a 2D plane with a constraint on the curvature of the path with predefined initial and final tangents [1]. These paths are assuming that the vehicle or system moves only in the forward direction. In this case, a simplified version of the Dubin's path is applied, where the goal heading is not defined.

A custom dataset will be developed using a provided MATLAB function that computes a Dubin's path given an initial position (x_1, y_1, and z_1), a goal position (x_2 and y_2), an initial heading (ψ), and a climb angle (γ). The dataset will be used to train three different RNN models to map the given initial and goal parameters to a trajectory. The trajectories from the three different models will be compared and evaluated.

Methods

The methodology will cover three areas: (1) data generation, (2) model development and hyperparameters, and (3) the training and testing scheme. Each of these sections and their implementation will be discussed in this section.

(1) Data Generation

For this application, a MATLAB function called "dubinEHF3d" was provided. The function computes a Dubin's path given an initial position (x_1, y_1, and z_1), a goal position (x_2 and y_2), an initial heading (ψ), and a climb angle (γ). The path itself is an Nx3 vector that contains the steps taken from x_1, y_1, and z_1 to x_2, y_2, and z_2. It should be noted that z_2 is not specified and is a function of the climb angle. An example path is shown in Figure 1.

In the example shown in Figure 1, it can be seen that the initial position for x_1, y_1, and z_1 is 0, and the final position of x_2 and y_2 are 100 and -250, respectively. For this case, an initial heading and climb angle of 20 degrees and 30 degrees, respectively, was used. Additionally, two other parameters defined the overall trajectory. First, the curve has a minimum turn radius of 100. Thus, any goal positions whose distance is less than the turn radius will create an invalid path. Additionally, a step length of 10 was utilized. This defined how long each step in the path is.

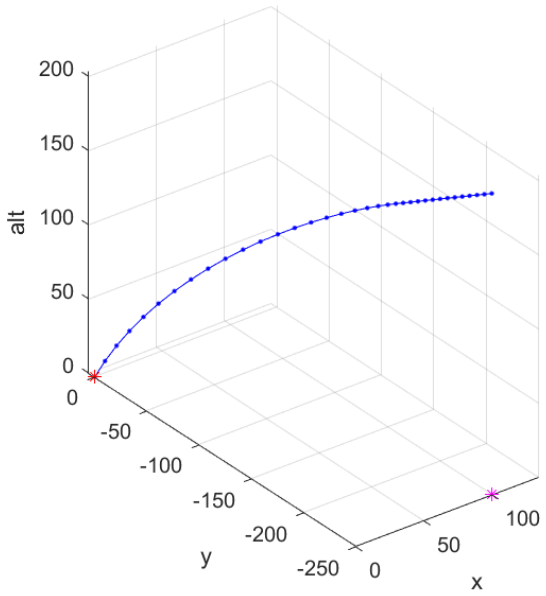

Figure 1: Example Dubin's path

Since PyTorch is being used to develop the model, the dataset needs to also be written in a Python structure. Thus, the provided script was rewritten in Python. To verify the implementation, the same inputs were provided to the Python dubinEHF3d function. Figure 2 visualizes the generated path.

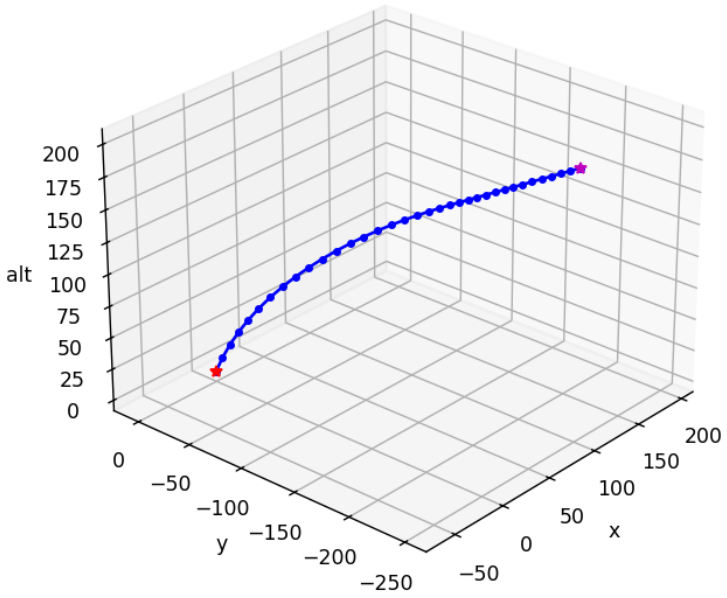

Figure 2: Example Dubin's path in Python

It can be seen that the path is identical.

To train a model to generate these trajectories, sequential training data, in the form of a trajectory, needs to be provided to the model. Thus, using the dubinEHF3d function, a dataset will be generated. For the dataset, the initial position of x_1, y_1, and z_1 for all the paths will be 0. For the goal position of x_2 and y_2, they will both range from -500 to 500, in intervals of 40. For the initial heading of ψ, it will range from 0 to 2π, in 30 degree intervals. For the climb angle of γ, it will range from -30 degrees to 30 degrees, in 5 degree intervals. These ranges were selected as smaller step sizes caused the computer to run out of memory during the data generation loop. Using different combinations of these parameters, a corresponding Dubin's path will be generated, and this will act as the ground truth data for the model. After generating this training data, the dataset will be padded. Since each path that is generated has a variable length, in order to pass the sequence to the model, the length of each path needs to contain the same number of points. Thus, each path will be padded with zeros to a length equal to the longest length sequence in the dataset. Once padded, the entire dataset can be normalized. In this case, the model will utilize a min-max normalization, only for the x_1 and y_1 input data and the x_2, y_2, and z_2 output data. The normalization range is between -1 to 1. This ensures that the values of the model are small enough to be trained without extremely high loss values.

After normalizing the dataset, the model can be split into training and testing datasets of 80% and 20%, respectively. From these datasets, a dataloader object can be created for the training and testing loop. The dataloaders will consist of a batch size of 32 and the training dataloader will be shuffled.

(2) Model Development and Hyperparameters

For this assignment, a neural network needs to be developed and trained to map the static initial conditions to a sequential output trajectory. To achieve this, a recurrent neural network (RNN) is needed. RNN's are known for sequential data processing. In an RNN model, there is a feature called a recurrent unit that maintains some form of memory. This memory is referred to as a hidden state. During each step of the sequence, the hidden state is updated with the current input and the previous hidden state. Using this feedback, the model learns from the data to generate a relationship over time [2]. For this application, three RNN architectures will be evaluated and compared: (1) a vanilla RNN, (2) a long short-term memory (LSTM) model, and (3) a gated recurrent unit (GRU) model. Each of these models incorporate a different architecture that will impact how the model learns the data. The results of the three models will be compared and evaluated.

A (1) vanilla RNN is a model that can be considered the simplest form of an RNN, where the hidden state is determined by the current input and the previous timestep. These types of

models are prone to vanishing and exploding gradients [3]. To apply this model, a torch.nn.RNN layer can be utilized in the model.

A (2) LSTM model is referred to as a long short-term memory model. The model is more complex than a vanilla RNN as it contains different features to capture short-term and long-term memory. It achieves this by incorporating three new gates into its hidden state, a forget gate, an input gate, and an output gate. These gates help prevent vanishing and exploding gradients and helps choose when to discard certain information [4]. To apply this model, a torch.nn.LSTM layer can be utilized in the model.

A (3) GRU model is referred to as a gated recurrent unit model. The model is similar to the LSTM architecture, however, it contains two gates versus three. The gates consist of a reset gate and an update gate, that are like the LSTM gating mechanism to input or forget certain features. However, there is no output gate as the LSTM has. The GRU's performance is similar to that of the LSTM, however, there is no clear answer as to which model is better [5]. The apply this model, a torch.nn.GRU layer can be utilized in the model.

One other aspect of the model development to consider is the RNN architecture to be used. For this application, there is a static input and a sequential output, thus, this problem will require a "one-to-many" architecture. Effectively, this means that "one" input will be used to predict "many" datapoints. This impacts how the model is implemented.

With these three models, certain hyperparameters need to be defined. First, the overall model architecture for each model will consist of a fully connected input layer that takes in 4 inputs and maps it to the size of the hidden layer, which in this case is 64. Various hidden layer sizes were tried, such as 16, 32, and 128. A size of 64 provided the best results. Additionally, since this model is a one-to-many architecture, the input commands are repeated for the length of the sequence. After the fully connected input layer, each model will have its own respective model layer, i.e. an RNN layer, an LSTM layer, and a GRU layer. For each of these layers, the layer will consist of 2 layers. Different layer sizes of 1 and 3 were tested but a size of 2 provided the best results and training time. Before and after each of these layers, there are packing and padding sequences that ensure the model does not train itself using the padded values of the dataset. Once the packed output is obtained, it is fed to a fully connected output layer that reduces the dimensions of the model from the hidden layer size of 64 to the output dimension of 3.

For these models, a mean squared error loss function is used to evaluate the loss. Additionally, an Adam optimizer with a weight decay of 0.01 is used with an initial learning rate of 1e-3, coupled with a learning rate scheduler. The scheduler reduces the learning

rate by a factor of 10 every three epochs, whenever the loss plateaus. Additionally, early stopping is implemented to prevent overfitting.

(3) Training and Testing Scheme

For each of the three models, the training loop is implemented as follows. For each batch in the training dataset, the inputs are passed to the model for obtaining predictions and subsequent losses are calculated. Using this loss, the optimizer calculates the gradients for the next step. This is repeated until all the batches are exhausted. Once the training is complete, the model switches to evaluation mode and the testing dataset loss is computed. After computing the test loss, the early stopping function checks whether the training needs to stop or not. If the training continues, the scheduler decides whether to lower the learning rate or not. This will mark the completion of an epoch and the loop will restart with this process. During the training loop, the training and validation loss will be logged via Tensorboard.

After the training is complete, the best model will be reloaded and the model will be used to make predictions on the test dataset. The predictions of the 10 inputs will be visualized and evaluated relative to the ground truth trajectory. Once complete, the model will be saved.

Results and Discussion

After developing the three models and the overall training and testing schemes, the models could be trained, tested, and compared. As mentioned, the model is ran with 50 epochs but will stop early as decided by the early stopping function.

The following plots depict the results of training and testing loss metrics of the three models. The three models and their line colors on the plots are:

- Vanilla RNN – Orange
- LSTM – Green
- GRU – Blue

Figure 3 visualizes the training loss of the three models.

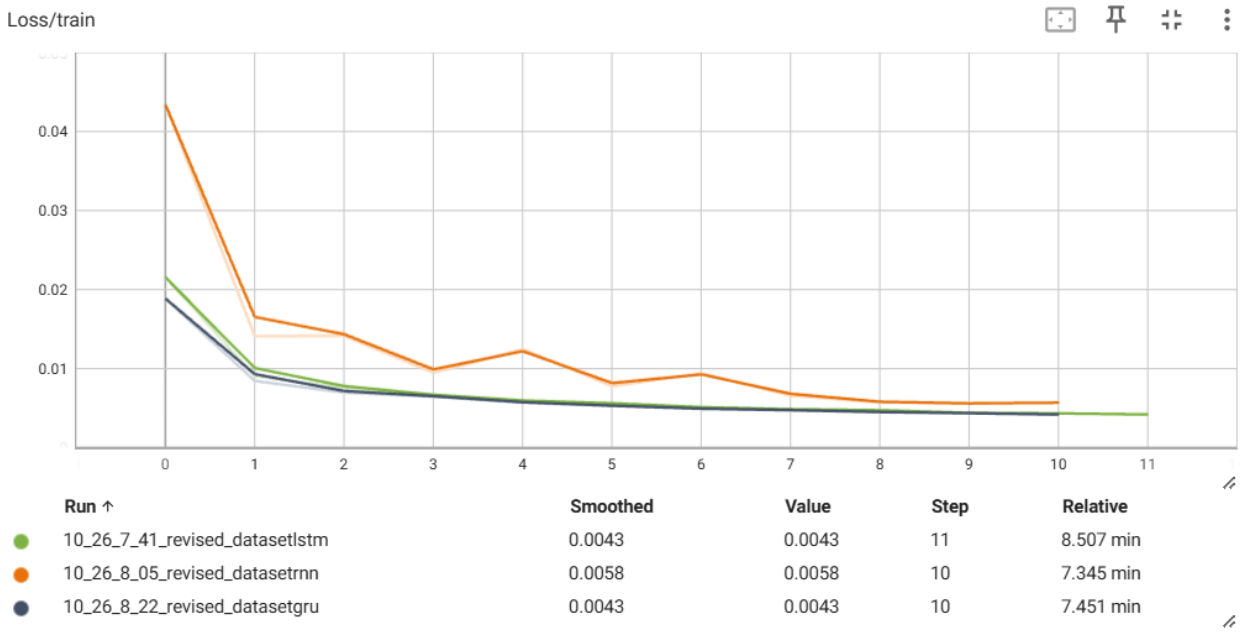

Figure 3: Training loss of the three RNN models

Figure 4 visualizes the testing loss of the three models.

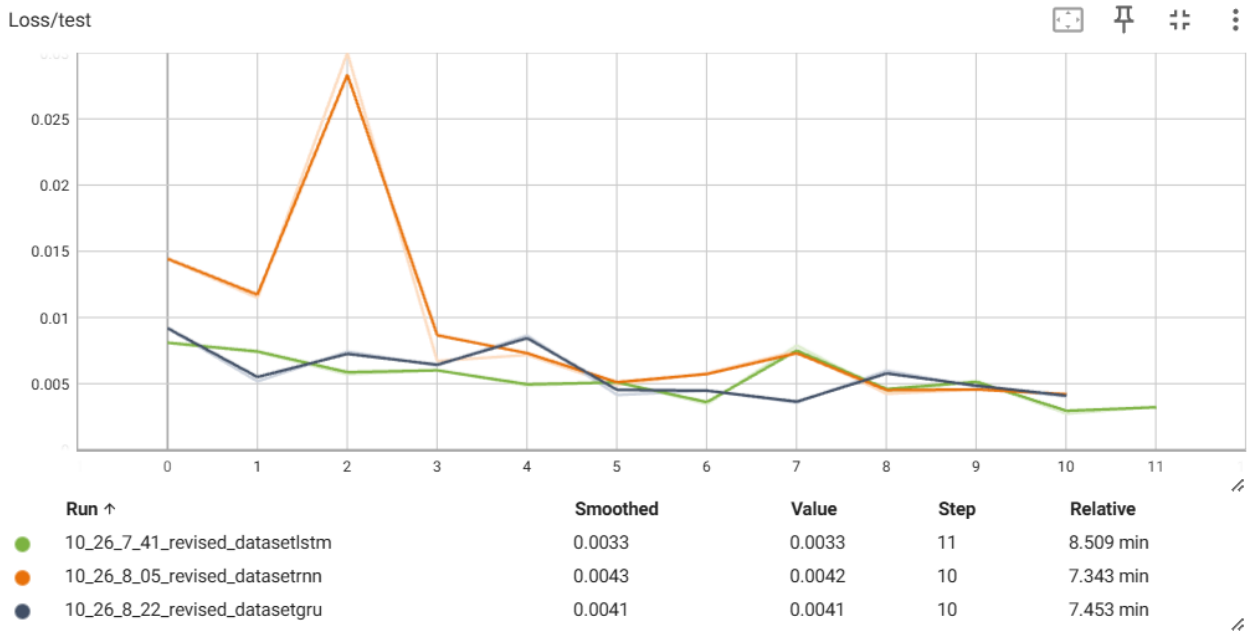

Figure 4: Testing loss of the three models

The best testing loss of the three models are the following:

- Vanilla RNN: 0.0049
- LSTM: 0.0035

- GRU: 0.0042

Based on Figure 3 and 4, the following observations can be made for each model

1. The vanilla RNN had the highest training loss across epoch.
2. The LSTM took the longest time to be trained and ended up with the lowest test loss.
3. The GRU followed a very similar training loss trend as the LSTM model. The model had the 2nd lowest test loss.

While testing loss is a good indicator about the model quality, additional investigation regarding the models trajectory generation will be critical. The following series of plots visualize the trajectories generated from the three different models for different test inputs.

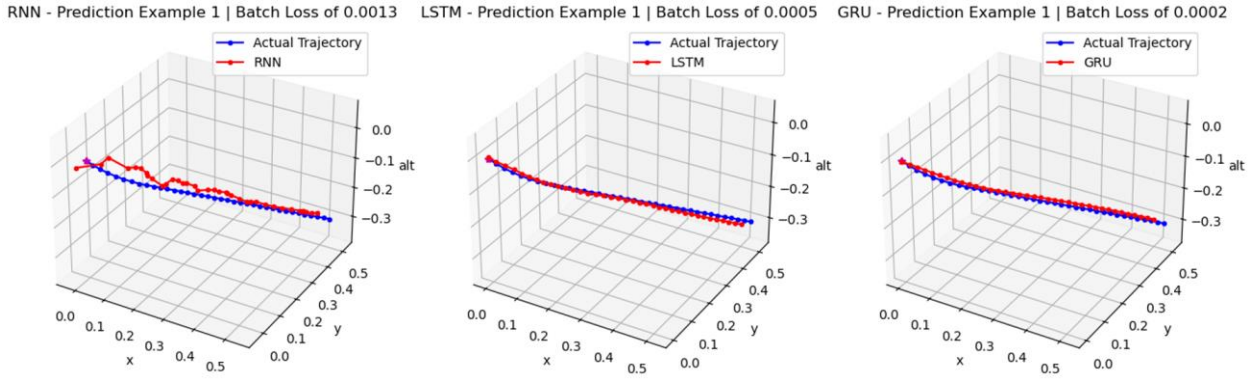

Figure 5: Trajectory predictions example 1

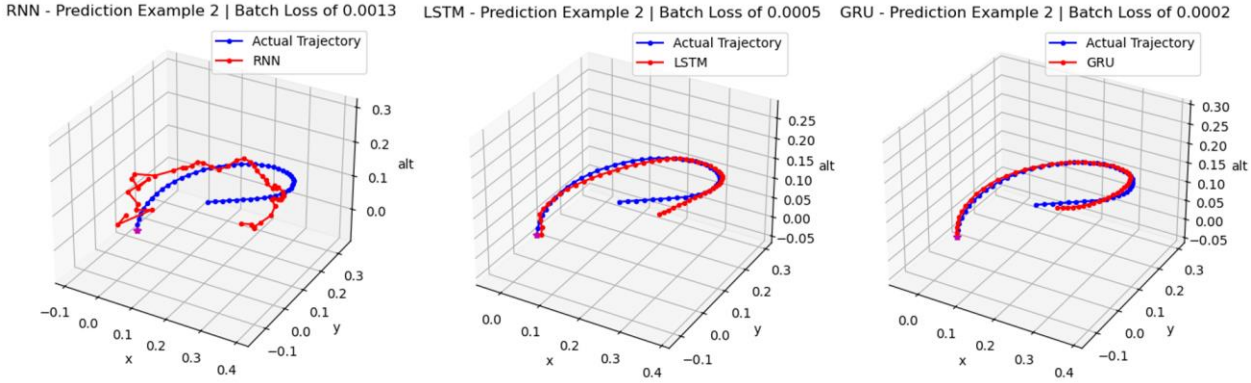

Figure 6: Trajectory predictions example 2

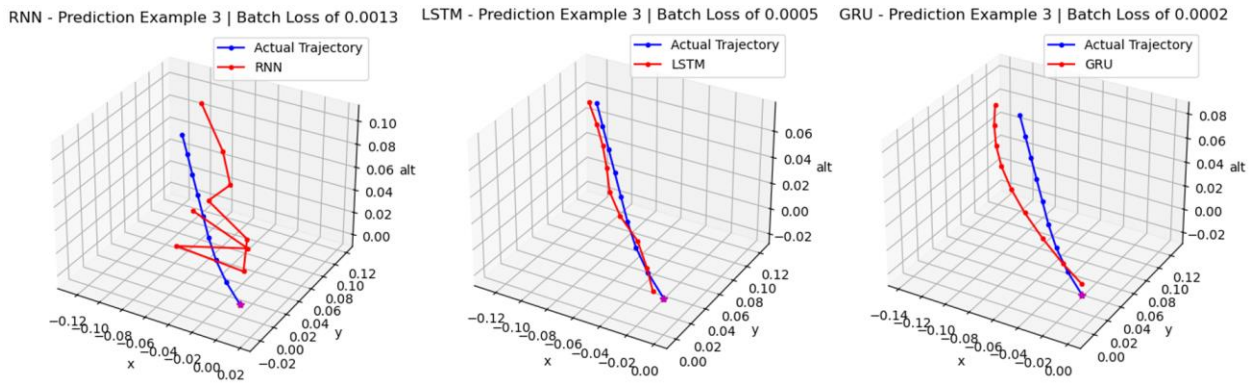

Figure 7: Trajectory prediction example 3

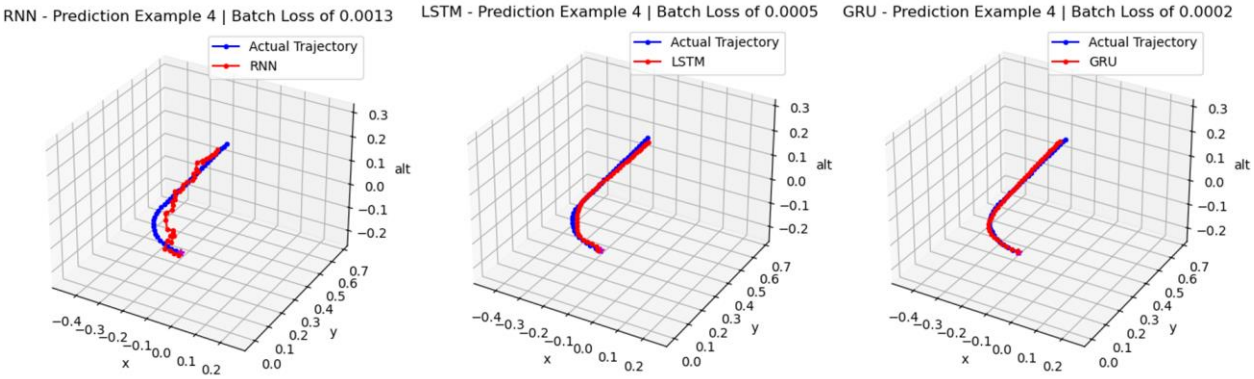

Figure 8: Trajectory prediction example 4

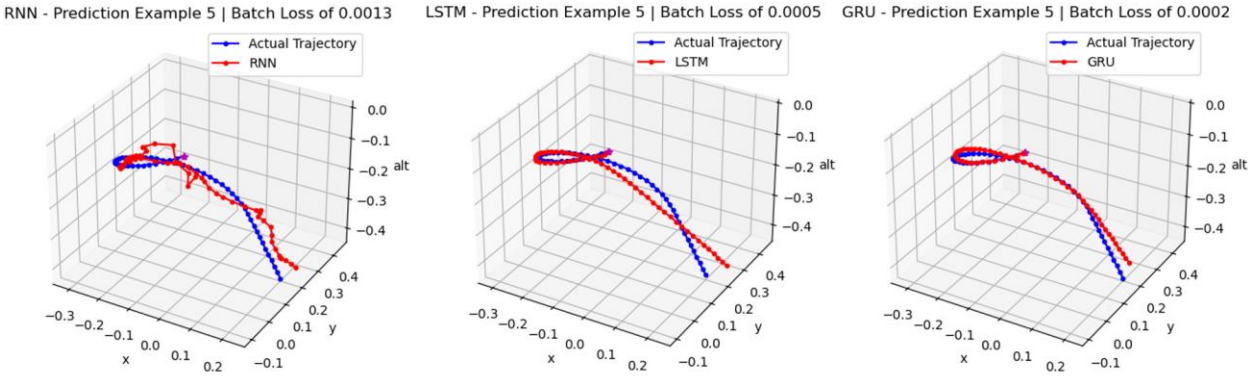

Figure 9: Trajectory prediction example 5

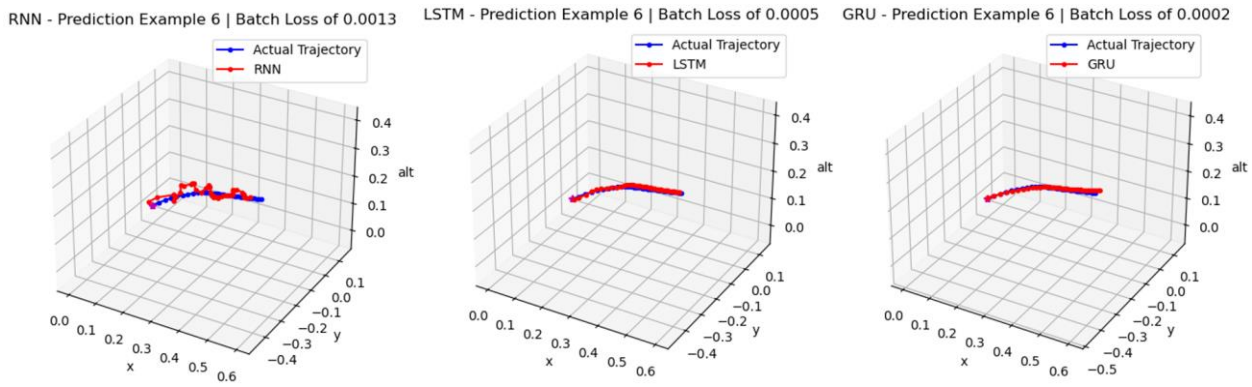

Figure 10: Trajectory prediction example 6

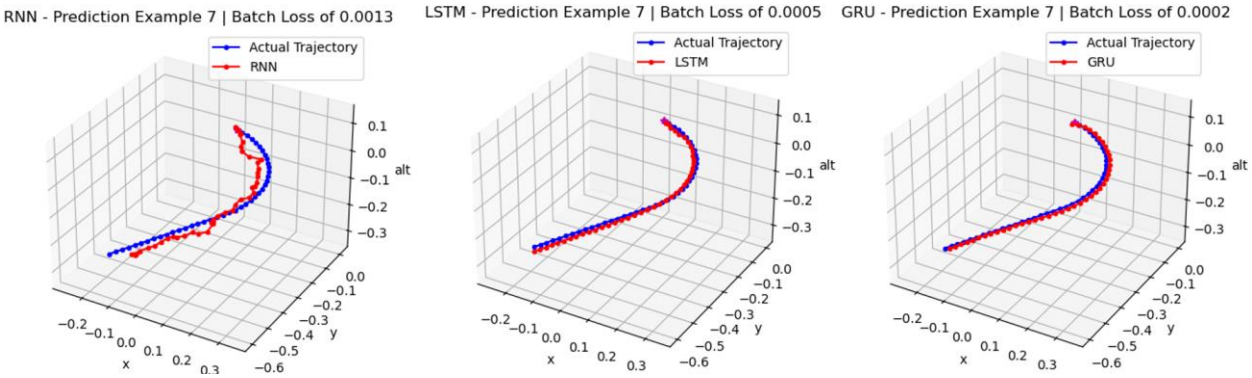

Figure 11: Trajectory prediction example 7

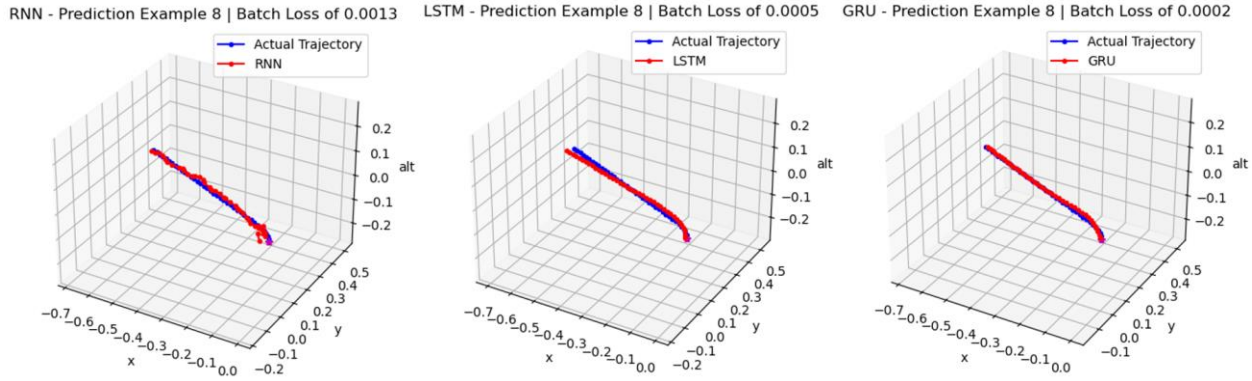

Figure 12: Trajectory prediction example 8

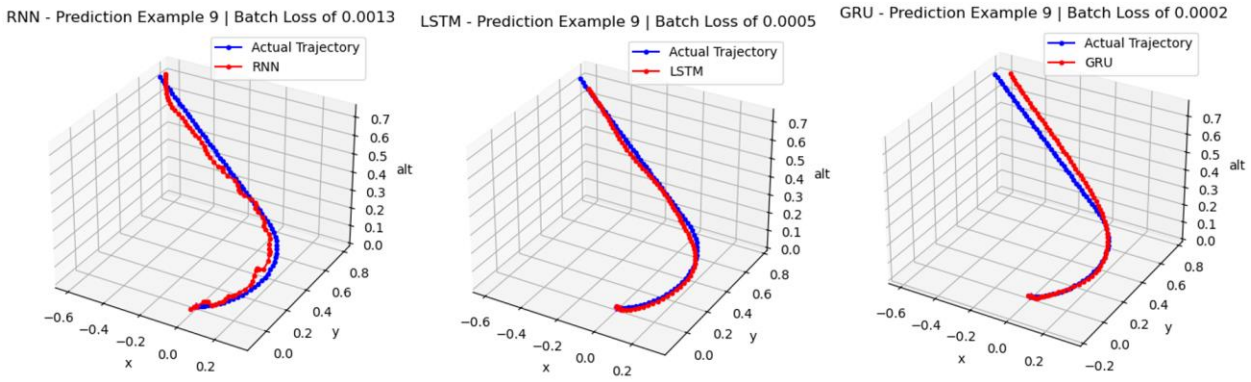

Figure 13: Trajectory prediction example 9

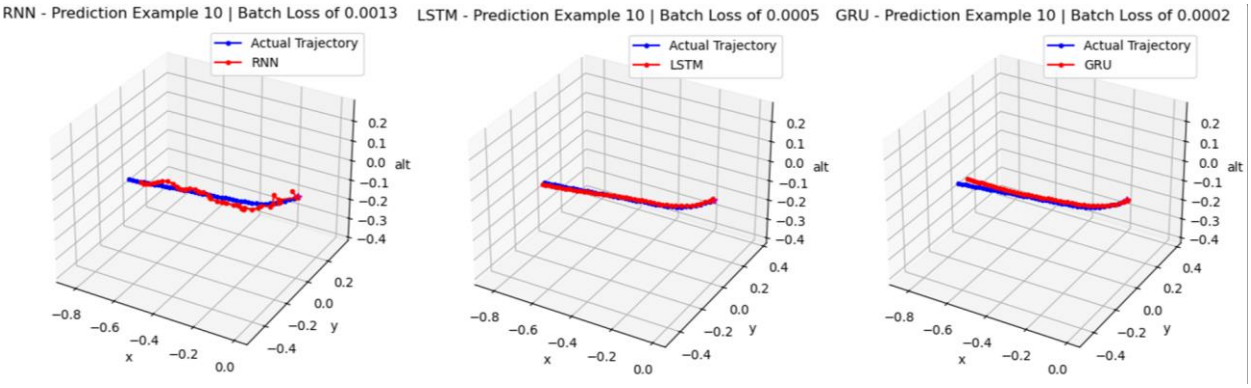

Figure 14: Trajectory prediction example 10

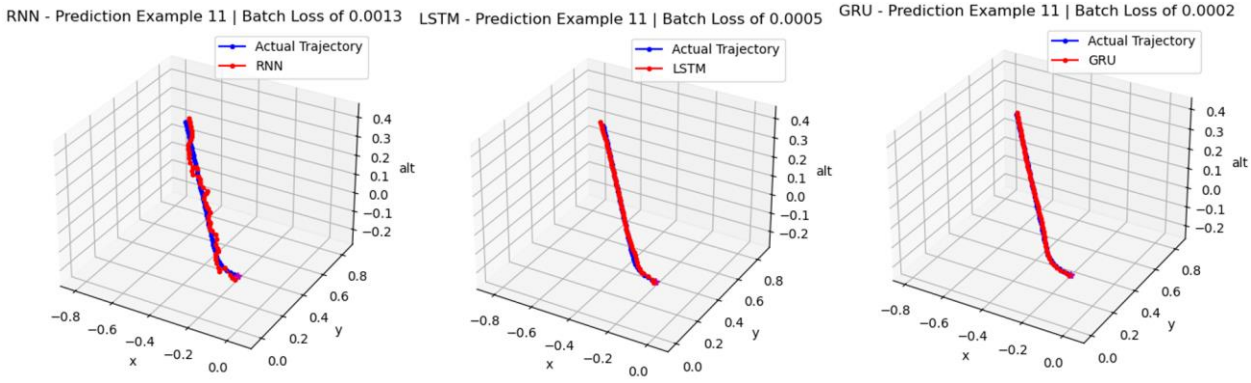

Figure 15: Trajectory prediction example 11

Based on Figures 5-15, the following observations can be made regarding the three models.

1. Vanilla RNN
    - The paths that are found are very inconsistent. For example, the prediction for test input 8 (Figure 12) follows ground truth relatively well, albeit not very

smooth. However, for test input 2, 3, and 5 (Figure 6, 7, and 9), the prediction is completely off.
    - It can be noted that the predictions struggle to follow straight or curved lines and tend to jump around the true path. This indicates that the model could not properly learn the Dubin's function and this is inherently caused by the usage of an RNN layer.
    - Even though the model had a relatively low training and testing error, the actual trajectories generated are not accurate.
2. LSTM
    - The paths that are found are relatively consistent with the ground truth. In many cases, such as test input 1, 4, and 11 (Figure 5, 8, and 15, respectively), the sequence followed the ground truth accurately. For other cases, such as test input 2, 3, and 5 (Figure 6, 7, and 9, respectively), the sequences were slightly off from the ground truth. However, even in these cases, the model was able to generate a sequence that had combinations of curved and straight paths.
    - Compared to the vanilla RNN, the LSTM model was able to predict and generate smooth, curved trajectories that are similar to that of a Dubin's path. This indicates that the model was successfully able to learn how a Dubin's path is generated.
    - The LSTM model had the lowest testing error, and the actual trajectories reflect this.
3. GRU
    - Similar to the LSTM model, the paths that are found are relatively consistent with the ground truth. In the same examples as the LSTM, for test inputs 1, 4, and 8 (Figure 5, 8, and 15, respectively), the sequence followed the ground truth accurately. For other cases, such as test input 2, 3, and 5 (Figure 6, 7, and 9, respectively), the sequences were slightly off from the ground truth. The predictions for these examples are an improvement over the LSTM predictions however. Additionally, even in these cases, the model was able to generate a sequence that had combinations of curved and straight paths.
    - Compared to the vanilla RNN, the GRU model was able to predict and generate smooth, curved trajectories that are similar to that of a Dubin's path. This indicates that the model was successfully able to learn how a Dubin's path is generated.
    - While the GRU model had a slightly higher loss than the LSTM, the predictions are very consistent with the ground truth.

When comparing the LSTM and GRU models, the trajectories that each model generates are very similar. In some cases, the LSTM model generates a better trajectory while in other cases, the GRU model generates a better trajectory. In both cases, the trajectories are consistently curved and/or straight, which indicates that the model has learned the inherent nature of a Dubin's path. As mentioned previously, the creators of the GRU architecture didn't come to a conclusion that a 2-gate architecture is better or worse than a 3-gate architecture. Based on the outcomes of this study, it is difficult to come to a clear conclusion on the same question. In both cases however, the LSTM and GRU outperformed the vanilla RNN counterpart by a significant margin.

There is greater room for improvement in the model outcomes. One major aspect is the dataset itself. If a larger dataset with smaller step sizes are feasible to be generated, it would theoretically improve the results of the model significantly. Another approach would be to tune or modify the overall design of the input, output, and RNN layers.

Overall, for this application, the different models were able to predict a Dubin's path, each with varying success. In a real-world application, it would be challenging to find a use case to select a data-driven model for generating Dubin's paths as compared to a mathematical model. For the context of this study, it provided an excellent framework for learning how to implement, train, and evaluate different RNN models.

References

[1] Wikipedia Contributors, "Dubins path," *Wikipedia*, Jan. 14, 2019. https://en.wikipedia.org/wiki/Dubins_path

[2] Wikipedia Contributors, "Recurrent neural network," *Wikipedia*, Dec. 03, 2018. https://en.wikipedia.org/wiki/Recurrent_neural_network

[3] P. Gaurav, "NLP: Zero To Hero [Part 2: Vanilla RNN, LSTM, GRU & Bi-Directional LSTM]," *Medium*, Mar. 23, 2023. https://medium.com/@prateekgaurav/nlp-zero-to-hero-part-2-vanilla-rnn-lstm-gru-bi-directional-lstm-77fd60fc0b44

[4] Wikipedia Contributors, "Long short-term memory," *Wikipedia*, Nov. 22, 2018. https://en.wikipedia.org/wiki/Long_short-term_memory

[5] Wikipedia Contributors, "Gated recurrent unit," *Wikipedia*, Feb. 18, 2019. https://en.wikipedia.org/wiki/Gated_recurrent_unit