UAV State Estimation Using IMU Vector with Deep Neural Networks

Rohit Alavala, Christian Micklisch, Sai Jasthi, Yuehan Lan

ITCS 5156 - Applied Machine Learning

Group 7

Introduction

In an autonomous system, knowing your current state is critical in determining the next decision. For an Unmanned Aerial Vehicle, UAV, the state refers to its position in an environment relative to its 6 degrees of freedom. A UAV will usually have an Inertial Measurement Unit, IMU, that helps track the linear acceleration and angular velocities utilizing an accelerometer and gyroscope. The linear and angular IMU readings can be stored as a vector where each index represents the readings of the IMU at a single moment in time. This vector would then be fed to a Deep Neural Network model which will determine the change in its state. If the model is provided with the original position of the UAV it can estimate the current state by providing a vector of all IMU readings from the original position to the current position.

In this paper, we specifically represent the UAV as a quadrotor, a drone with 4 propellers equidistantly in space to allow for movement in the Euler space. We represent the 3 rotational positions of roll, pitch, and yaw as phi, theta, and psi respectively. The 3 translational positions are x, y, and z.

Problem Statement

While there exist state estimators for UAVs in simulated environments few rely on strictly the raw data produced by IMUs. IMU's represent a mixture of sensors utilized to help determine the linear accelerations, angular velocities, and heading. The linear accelerations are provided by the accelerometer, the angular velocities are handled by the gyroscope, and the heading is handled by a magnetometer. The state of a UAV contains the linear and angular positions, relative to where it started. The linear position is the x, y, and z position along a 3D grid where x represents the distance, y represents depth, and z represents height. The angular position is the ϕ, Θ, and Ψ. The diagram below shows what the state would be for the linear and angular positions.

For state estimation, there is also a subsection, attitude estimation, that focuses on only determining the angular positions, e.g. ϕ, Θ, and Ψ. In this paper, we will refer to ϕ, Θ, and Ψ as phi, theta, and psi respectively.

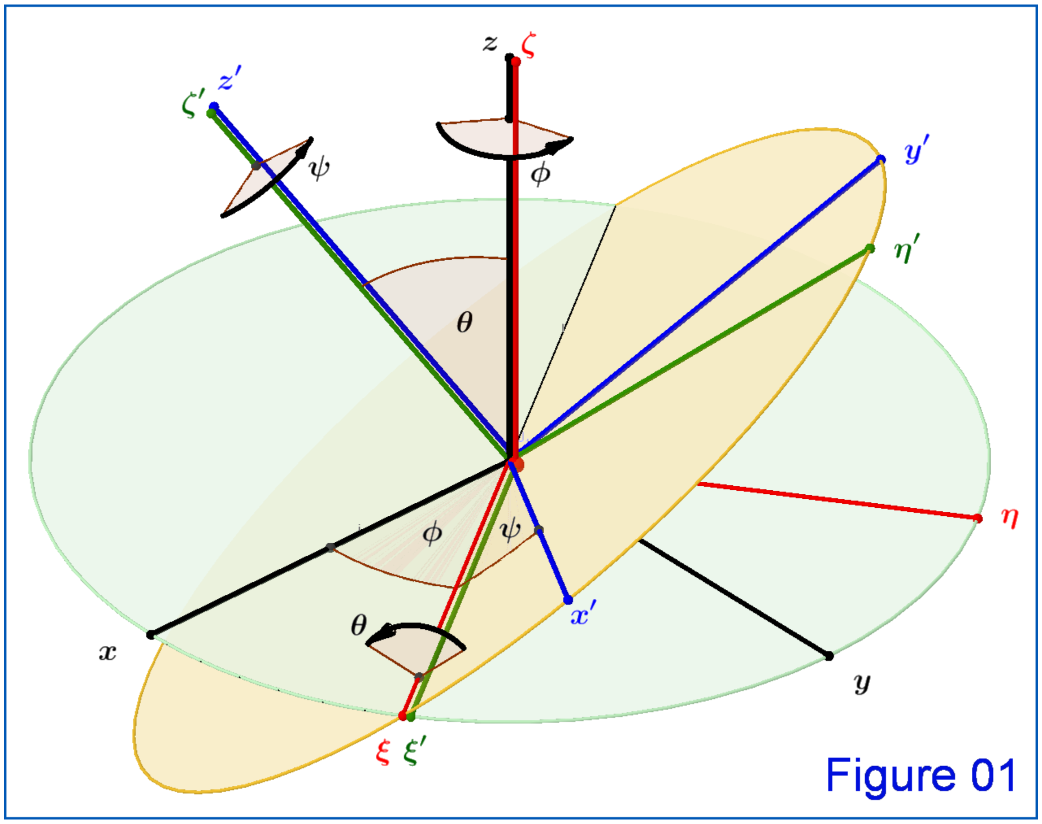


Figure 01

The IMU outputs have a lot of sensor noise associated with it, which makes it difficult to write a set of equations that will describe the attitude of the UAV directly from the outputs of the accelerometer and gyroscope. Currently, a Kalman Filter can be utilized to digitally filter out the noise of the sensors. The output of the filtered data can then help estimate the attitude of a UAV's position, which can then be utilized to estimate the state. This process of using a Kalman Filter to determine the state is called the Extended Kalman Filter. (Yang et al.)

A survey of the literature of UAV state estimation shows that most state estimators that utilize machine learning models, rely on/off-board optical sensors to train the model for state estimation. Currently, the only research on IMU sensor usage for UAVs is Al-Sharman et al., they created a self-written DLNN to replace the Kalman filter to help estimate the attitude, finding that the DLNN is more accurate than the Kalman Filter. The results of Al-Sharman et al.'s DLNN are shown below.

Our goal is to create a model that can estimate the state of the UAV from the raw data using a generalized model of the IMU error dynamics. This model would act as an Extended Kalman Filter for any IMU sensor.

	DLNN RMSEE
phi	0.00016
theta	0.00023
psi	0.00200

When relying on IMU sensor data to train the model one issue is that it's dependent on the sensor we select. Each sensor has its error dynamics wherein the noisy output may be directed towards a specific average value or the output has a certain Gaussian noise. This means that were we to train our model on a specific IMU, then the model might only work on the specific IMU. Another issue that represents itself is there is currently no data published by Al-Sharman et al. nor DLNN code, that can be utilized to help train a model. This produces 2 main challenges:

1. The sensor that we train the model on determines the sensor that we need to utilize when feeding data to the model

2. There is no data available to recreate Al-Sharman et al.'s model, let alone train our model.

Motivation

One of the most difficult areas of designing a UAV is state estimation, especially with common off-the-shelf components. Current commercial IMU's are cheaply available but contain sensor error dynamics such as noise and drift. (Nirmal, K., et al) These error dynamics are usually resolved by the use of an extended Kalman filter (EKF), reviewed by Yang et al., but are complex to implement and require background knowledge of the Inertial body of the UAV. Al-Sharman et al. replaced the Kalman filter and fed the raw IMU data directly to the ML model. This resulted in an RMSEE score for the angular positions' phi, theta, and psi of 0.00016, 0.00023, and 0.00200 respectively. We want to see if we can improve the accuracy of this model by sending the raw IMU data as a vector of recorded readings over a time period and predict not only the attitude, angular position but the linear position as well. The state we are trying to estimate is the angular position and linear position of a quadrotor drone every 0.03 seconds.

Approach

To resolve both issues, we need to somehow create data that will reflect the real state of the UAV, while also providing the current IMU readings of the accelerometer and gyroscope. This data also needs to represent multiple different sensors, with a broader range of error dynamics that can reflect varying IMUs. The data also needs to represent multiple different flights to allow various error dynamics of the IMU to show themselves. To do this we utilize Micklisch et al.'s simulator to generate the data, which utilizes simulated sensors to comprise an IMU with broad error dynamics. The simulator can be run to output a CSV that represents a single flight path which will contain the real state of the UAV in the simulated environment along with the accelerometer and gyroscope readings of the IMU. This data is pulled during the flight every 0.03 seconds. Utilizing this simulator we can produce hundreds of random flight path CSVs which can then be preprocessed into a single CSV to train the model. (This data would be simulated and self-made)

Micklisch et al.'s simulator, used to produce the training data, was originally implemented to test various controlling algorithms utilized for autonomous quadrotors. For autonomous flight, a quadrotor will determine autonomously how to reach a given position utilizing a takeoff/landing controller, attitude controller, and collision detection. The takeoff/landing controller helps determine the speed at which to start-up/stop and the height at which to stabilize itself before heading towards the given position. The attitude controller determines the angle at which the quadrotor should be, for it to move towards the given position. The obstacle detection method should prevent the quadrotor from crashing into a wall and direct the quadrotor around an obstacle to reaching the given position safely. Each of the components that a quadrotor utilizes to move and detect was recreated inside the simulator. The inertial structure of the quadrotor, motor control systems, propeller shape, are also variable, allowing any current production quadrotor to be recreated. The IMU, Ultrasonic, and pressure sensors are also variable allowing for multiple different error dynamics, inertial structures and refresh rates to be set.

Related Work

As described by Yang et al. traditional State Estimation can be done using three core techniques, attitude estimation via an Inertial Measurement Unit (IMU), Pose estimation via extended Kalman filtering, and optimal state estimation. Outside of traditional works in state estimation Machine learning methods have been derived. Carrio et al. touched on the use of DNN and CNN for motion control with low-level control policies from imperfect optical sensor data in simulations implemented by Aznar et al. and Ross Girshick respectively. Jaradat and Hafez explored an integrated navigation solution, INS, that uses IMU sensors to provide data to a Neural Network which has been pre-trained with the sensors' error dynamics. This INS system characterizes the x, y, and z vector spaces individually. The INS was initially tested in a simulated environment and then run against real GPS/IMU data. Cheon et al. utilized Proportional Integral Derivative, PID, controller outputs to train a Deep Learning Controller on how to control a DC Motor System. Ibarra-Bonilla et al. developed a Kalman Filter with Neuro-Fuzzy logic for attitude estimation using the raw data of an IMU. Jin et al. proposed a relational graph network derived from a relational neural network to accurately predict a 6D pose of a quadrotor from optical sensors. Al-Sharman et al. implemented a deep learning framework to improve state estimation by identifying measurement noise and filtering it out.

Ibarra-Bonilla et. al's attitude estimation was only utilized with Pedestrian Dead Reckoning, PDR, and didn't compare the Neuro-Fuzzy adaptation of the Kalman Filter with the current UAV flight. The use of state estimation from optical sensors by Jin et al., Aznar et al., and Ross Girshick may be ineffective in foggy or low light situations. Jaradat and Hafez's use of GPS fusion will limit their Navigation System in Urban and indoor environments.

The pros are: a deep neural network (DNN) can ensure identifying representative, fitted, and robust models for the unfiltered measurement noise. The deep-learning-based neural network (DLNN) attitude estimation results exhibit superiority over the conventional attitude estimates. The corns are: the usage of DNN and DLNN models may only learn the error dynamics of the sensors it was trained against, limiting the models robustness against other sensors.

Neural networks are known to have non-convex loss function surfaces, and often handle very high-dimensional data, which adds to the complexity of their analysis and makes sound theoretical results difficult to achieve. Furthermore, there does not exist a standard and compact algebraic framework for neural network researchers to operate within(Caterini, 2018). Deep neural networks are capable of discovering latent structures (or feature learning) from unlabeled and unstructured data, such as images (pixel data), documents (text data), or files (audio, video data). What differentiates any deep neural network from an ordinary artificial neural network is the way we use backpropagation. In an ordinary artificial neural network, backpropagation trains later (or end) layers more efficiently than it trains initial (or former) layers. Thus, as we travel back into the network, errors become smaller and more diffused (Goyal, 2018).

In the future, we can try to use RNN. A Recurrent Neural Network (RNN) is a type of neural network architecture which has a loop pointing to the same circuit, unlike the normal feedforward neural networks (Bableshwar, 2019). The main idea behind RNNs is to use sequential information. In other neural networks, all inputs and outputs are independent of each other. But for many tasks it does not work well. For example, If you want to predict the next word in a sentence, it is better to know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, and the output depends on the previous computations. Also we can say that RNNs have a "`memory"' which captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps (Deshpande, 2018).

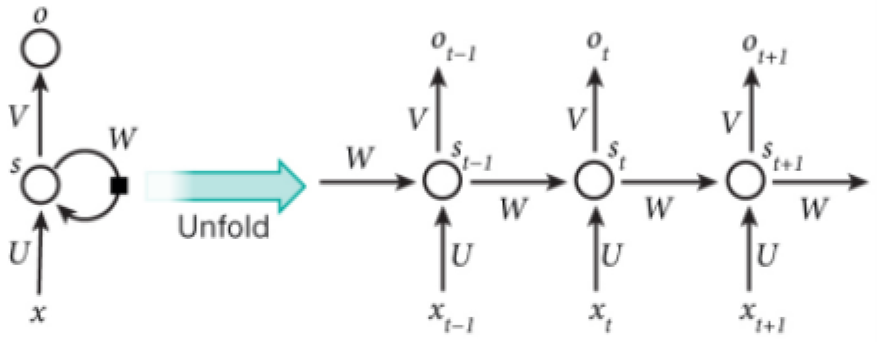


figure 1: recurrent neural network and the unfolding in time of the computation involved in its forward computation (Deshpande, 2018).

Method

Our methodologie starts with discussing how the data was generated from the simulator and its structure, covered in "Data Generation". In "Preprocessing" we discuss how the data generated is altered to be in a vector structure and how the target values for the training and testing phases are generated. In "State Estimation Structure" we review how the models we trained are then utilized to determine the state from an entire flight from the data generated. In "Training Models" we skim through how the models were trained, what our different model structures are and the hyperparameters we select.

Data Generation

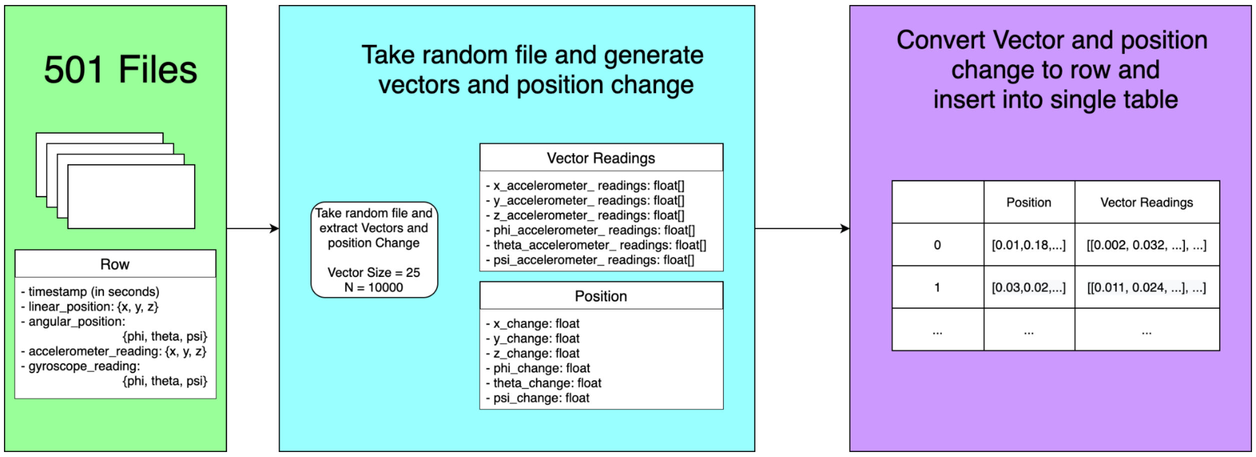
To generate the data Micklisch et al.'s simulator was utilized to run several hundred flights. In each flight the simulator outputted the current state of the drone and its sensor readings every 0.03 seconds. To run the simulator and generate various flights, a bash script was written to trigger the simulator to run and capture all the printed data into a csv file. The table below describes all the data files. Each "desired_linear_position_*" was randomly set in the beginning of the flight. Currently the specific Attitude Controller was set to Kempers PID controller. White noise was used for the error dynamic of the IMU. This allows error dynamics to be as broad as possible with sensor readings fluctuating randomly given the quadrotors specific state. In total around 501 flights were generated. Each flight is stored in the "data" directory as a timestamp csv.

Column Name	Description
timestamp	Refers to the amount of time that has passed since the start of the simulation. Time passes at indexes of 0.03 seconds.
attitude_controller_id	Refers to the attitude controller that is utilized during the flight. All flights should use the Kemper, "k", attitude controller. The attitude controller defines how the quadrotor is going to fly
desired_linear_position_x	The target position that the drone wants to be at in the x direction
desired_linear_position_y	The target position that the drone wants to be at in the y direction
desired_linear_position_z	The target position that the drone wants to be at in the z direction
linear_position_x	The actual x linear position of the drone in the simulated environment.
linear_position_y	The actual y linear position of the drone in the simulated environment.
linear_position_z	The actual z linear position of the drone in the simulated environment.
angular_position_phi	The actual ϕ angular position of the drone in the simulated environment.
angular_position_theta	The actual θ angular position of the drone in the simulated environment.
angular_position_psi	The actual ψ angular position of the drone in the simulated environment.
accelerometer_reading_x	The reading of the accelerometer attached to the drone from the x direction.
accelerometer_reading_y	The reading of the accelerometer attached to the drone from the y direction.
accelerometer_reading_z	The reading of the accelerometer attached to the drone from the z direction.

gyroscope_reading_phi	The reading of the gyroscope attached to the drone from the ϕ direction.
gyroscope_reading_theta	The reading of the gyroscope attached to the drone from the θ direction.
gyroscope_reading_psi	The reading of the gyroscope attached to the drone from the ψ direction.

Pre processing

Since our data is scattered through 501 different files with several thousands of rows in each one, we want to create a dataframe that will describe the change in the x, y, z, phi, theta, and psi from a vector of readings of the accelerometer and gyroscope. To get the vector information together, with the change of the angular/linear positions, a function was written to get the data of a random file, get a random sequential list of rows in the file, and get the difference in time and positions along with a vector of the readings starting at the beginning row. The figure below shows an overview of the entire preprocessing phase.


501 Files
Row
- timestamp (in seconds)
- linear_position: {x, y, z}
- angular_position: {phi, theta, psi}
- accelerometer_reading: {x, y, z}
- gyroscope_reading: {phi, theta, psi}
Take random file and generate vectors and position change
Take random file and extract Vectors and position Change
Vector Size = 25
N = 10000
Vector Readings
- x_accelerometer_ readings: float[]
- y_accelerometer_ readings: float[]
- z_accelerometer_ readings: float[]
- phi_accelerometer_ readings: float[]
- theta_accelerometer_ readings: float[]
- psi_accelerometer_ readings: float[]
Position
- x_change: float
- y_change: float
- z_change: float
- phi_change: float
- theta_change: float
- psi_change: float
Convert Vector and position change to row and insert into single table
Position
Vector Readings
0
[0.01,0.18,...]
[[0.002, 0.032, ...], ...]
1
[0.03,0.02,...]
[[0.011, 0.024, ...], ...]
...

To get a random file, we load all the files in the "data" directory as a list, select a random file name from the list, and load the file as a pandas DataFrame. Each file has the CSV headers in the first line.

To get a random list of sequential rows we select a random starting position and determine the end position by adding the start position to the total amount of rows we want to pick. The list of rows from a random starting position (minus one) to the end position (starting position + amount). The difference of the quadrotors position needs to be calculated from at least 2 rows, as you need the first row to provide insight into the readings of the quadrotor (IMU readings) and the second row to see where the quadrotor ended up (linear and angular positions). The function to get a random list of rows is shown below.

```
get_random_rows(df, amount):
   if (df.shape[0] - amount <= 1):
       return pd.DataFrame()
   start_index = random.randint(1, df.shape[0] - amount)
   return df.iloc[(start_index - 1):(amount + start_index), :]
```

To get the difference in the start and end positions, along with a vector of the IMU readings, we utilize the randomly selected rows. These rows are a DataFrame, which allows us to get the difference in values from the first to the last element for all of the state positions and create the vector of all the IMU readings by utilizing the "iloc" array positions. The code for converting the random rows selection to a single row for the training dataset is shown below.

```
convert_to_table_row(df):
   try:
       row = [
           df["timestamp"].iloc[-1] - df["timestamp"].iloc[0],
           df["linear_position_x"].iloc[-1] -
               df["linear_position_x"].iloc[0],
           df["linear_position_y"].iloc[-1] -
               df["linear_position_y"].iloc[0],
           df["linear_position_z"].iloc[-1] -
               df["linear_position_z"].iloc[0],
           df["angular_position_phi"].iloc[-1] -
               df["angular_position_phi"].iloc[0],
           df["angular_position_theta"].iloc[-1] -
               df["angular_position_theta"].iloc[0],
           df["angular_position_psi"].iloc[-1] -
               df["angular_position_psi"].iloc[0]
       ];
       row += df["accelerometer_reading_x"].iloc[:-1].tolist()
       row += df["accelerometer_reading_y"].iloc[:-1].tolist()
       row += df["accelerometer_reading_z"].iloc[:-1].tolist()
       row += df["gyroscope_reading_phi"].iloc[:-1].tolist()
       row += df["gyroscope_reading_theta"].iloc[:-1].tolist()
       row += df["gyroscope_reading_psi"].iloc[:-1].tolist()
       return row
   except IndexError as error:
       print(df + " failed");
       return False
```

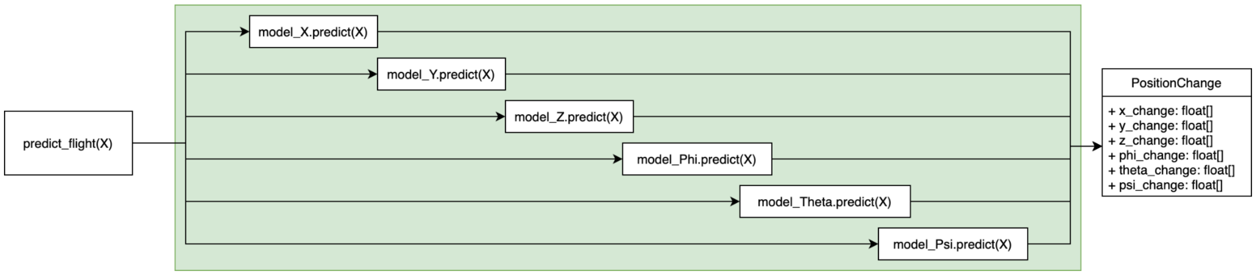
To create our training table we simply determine the amount of randomly selected rows and the amount of rows we want in our training table. We then select a random file and get the rows until our table is filled. The code to generate the training table is shown below.

```
    generate_table(files, vector_size, row_size):
        df_first = get_dataframe(files[0]);
        columns = df_first.columns
        table = []
        for i in range(row_size):
            random_rows = get_random_rows(
                read_random_file(files, columns), vector_size)
            if (random_rows.empty):
                continue
            row = convert_to_table_row(random_rows)
            if (row == False):
                continue
            table.append(row)
        return pd.DataFrame(table,
            columns=generate_columns(vector_size))
```

From our "generate_table" function, we generate a set of tables from a selection of row sizes and vector sizes. Our row sizes are 2000, 4000, 6000, 8000, and 10000. Our vector sizes are 25, 100, 225, 450, 600, and 900. From quick preliminary testing we determined that the vector sizes of 25 and 100 produced the best RMSE scores and the row sizes themselves provided negligible differences in RMSE performances. As such, for quick training purposes we compared models utilizing the tables with a vector size of 25 and 100 with a row size of 2000. For final training we utilized the table with a vector size of 25 and a row size of 10000. In the "notebooks/1-preprocessing" jupyter notebook, the preprocessing is performed. The results, the generated tables, are stored in the "processed_data" root directory.

State Estimation Structure

To help determine the state of the quadrotor we needed to create a State Estimator class that contained all of the models and determined the output of the predictions from the flight path file. The overall structure of the state estimation utilized 6 different models to determine the 6 different positions of the quadrotor in flight. The below image is an example of how the state was estimated from an entire flight. In our code, we train our models separately and save them in the "Models" directory. From there we load the models into the State Estimator class to then predict our testing and flight path data.



The data, X, needs to be created by transforming the original flight file by utilizing an altered "generate_table" method, shown below. This method produces an input dataframe with the same structure as the training data set generated in the preprocessing step. When passed to the state estimators predict_flight method the predicted flight is represented as a selection of position changes, shown in the image above. Each of these position changes can be summed together to represent the total state of the flight.

```
generate_table(df, vector_size):
    table = []
    for i in range(df.shape[0] - vector_size):
        random_rows = get_index_rows(df, i, vector_size)
        if (random_rows.empty):
            continue
        row = convert_to_table_row(random_rows)
        if (row == False):
            continue
        table.append(row)
    return pd.DataFrame(table,
        columns=generate_columns(vector_size))

get_index_rows(df, start_index, amount):
    if (df.shape[0] - amount <= 1):
        return pd.DataFrame()
    return df.iloc[
        (start_index):(amount + (start_index + 1)), :]
```

To generate the prediction of the state of the entire flight, we utilized the code below. Since each vector is offset by a 0.03 second time, we can simply add our current change in position to the overall change that has occurred and we will get our current position. For our problem, we know that, if we add the current change to the previous change, we get the total change between the two data points, hence in our "getFlightPrediction" we store the first change in out "time_prediction" array, and then add the next time change to the array with the previous "time_prediction" index value.

```
getFlightPrediction(df, position_changes):
    time_prediction = []
    index = 0;
    for change in position_changes:
        if (index == 0):
            time_prediction.append(change[0])
        else:
            time_prediction.append(
                time_prediction[index - 1] + change[0])
        index = index + 1
    return time_prediction
```

In the "notebooks/7-estimate-flight" jupyter notebook, the State Estimator is utilized to determine the flight path for several flights and loads in several models from the "Model directory.

Training Models

For our approach we trained separate models to predict each of our targets individually, this gave a significant performance boost. We tried hundreds of different DNN sequential layer structures, such as used Keras libraries We tried as many activations as possible, such as Sigmoid, Softmax, Softplus, Softsign, Tanh, Selu, and Elu. We added different types of normalization layers such as: LayerNormalization, and BatchNormalization. Dropout layers were also experimented with Different epochs, and batch_size were also experimented with.

Our method for tracking model performance was done manually, each model result was saved as a screen shot, in the "Notebook_screenshots" root directory of the github repository.

Experiments

The table below goes through every experiment we did.

Targets	Optimizer	Learning rate	Activation	Epochs	Batch Size	Dropout
x_change y_change z_change phi_change theta_change psi_change	sgd	0.01, 0.001, 0.0001	selu,elu, relu, sigmoid, softmax, softplus, Softsign, tanh	30-350	6,12	0.1-0.9
Same as above	adam	0.001	Same as above	30-350	6,12	Same as above
Same as above	RMSprop	0.001	Same as above	30-350	6,12	Same as above
Same as above	Adagrad	0.001	Same as above	30-350	6,12	Same as above
Same as above	Adadelta	0.001	Same as above	30-350	6,12	Same as above

The table below shows what worked best in relation to what the targets are.

Target	Optimizer	Learning rate	Activation	Epochs	Batch Size	Dropout
x_change y_change	adam	0.001	elu	150	6	0.32, 0.25, 0.1
z_change	adam	0.001	elu	250	6	0.32, 0.25, 0.1
phi_change theta_change psi_change	sgd	0.01	sigmoid	100	12	0.32, 0.25, 0.1

The table below shows our best metrics/results achieved for each target.

Target	R2	max_err	MAE	MAPE	MSE	RMSE
x	1.65e-01	1.83e-01	3.44e-02	5.93e+03	3.50e-03	1.22e-05
y	1.29e-01	1.83e-01	3.29e-02	1.79e+03	3.17e-03	1.01e-05
z	-4.07e-01	2.96e+00	9.61e-02	6.63e+02	1.14e-01	1.30e-02

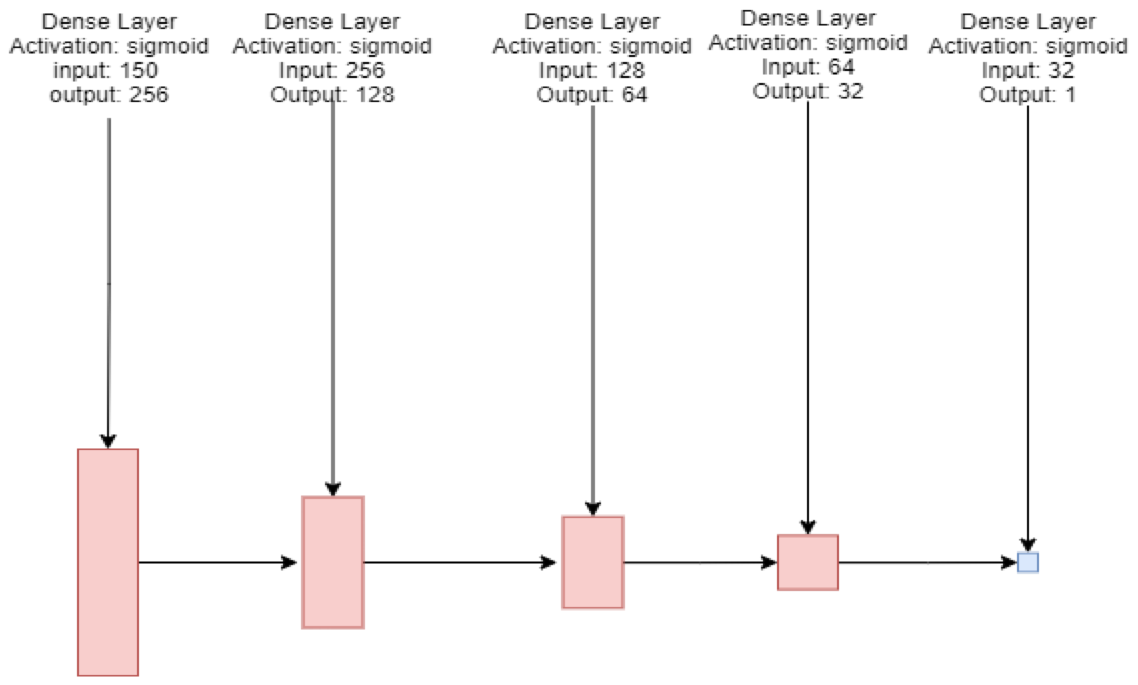
phi	-1.68e+00	1.59e-02	8.75e-04	9.45e+05	1.13e-06	1.29e-12
theta	-9.32e-01	1.09e-02	6.36e-04	2.26e+05	7.18e-07	5.15e-13

This table shows Al Sharman et al. Results.

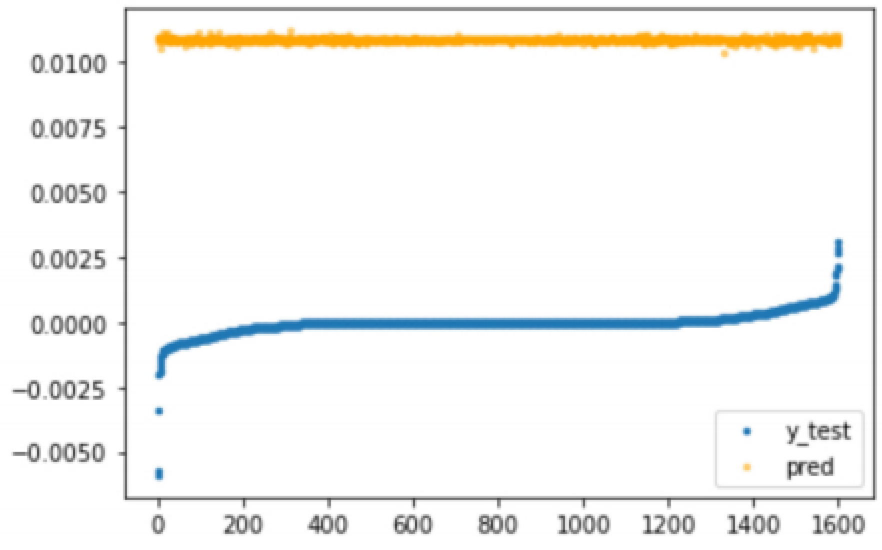
	DLNN RMSEE
phi	0.00016
theta	0.00023
psi	0.00200

Explanation of Experimental Setup

The first step of our experimentation was to find the ideal number of Dense layers to have. This step was fairly straightforward because we knew how many features we had which was 150 features. Using this information we got our basic sequential structure. This was our first most basic structure.



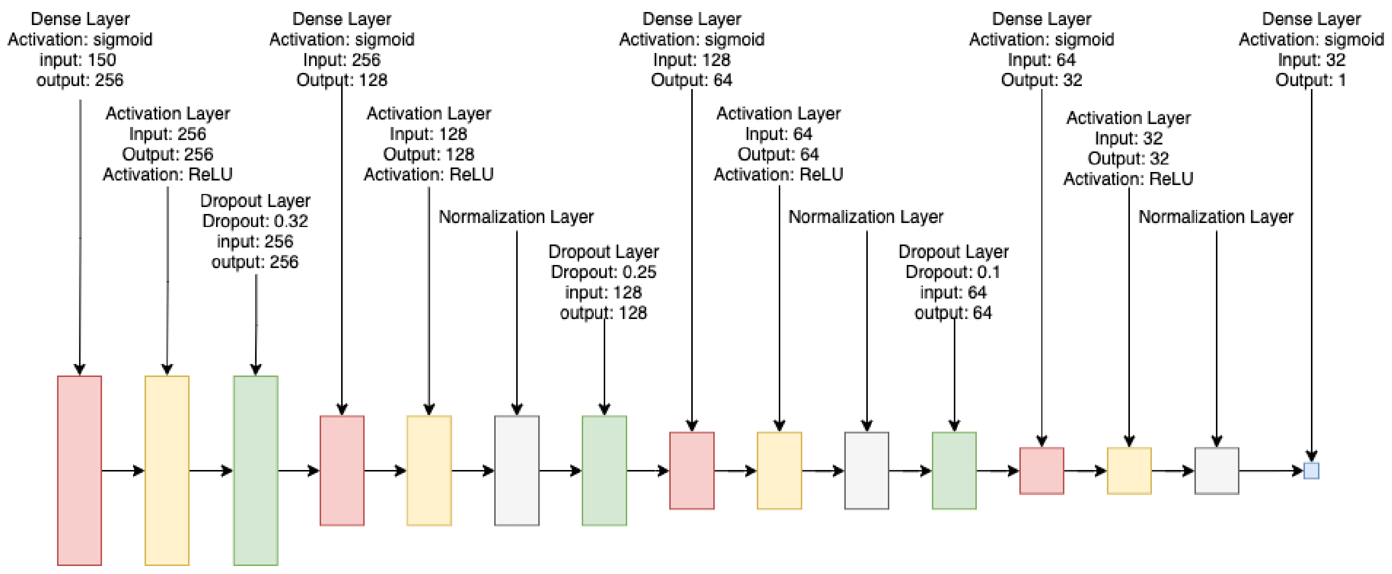
These were our results for the above DNN structure.



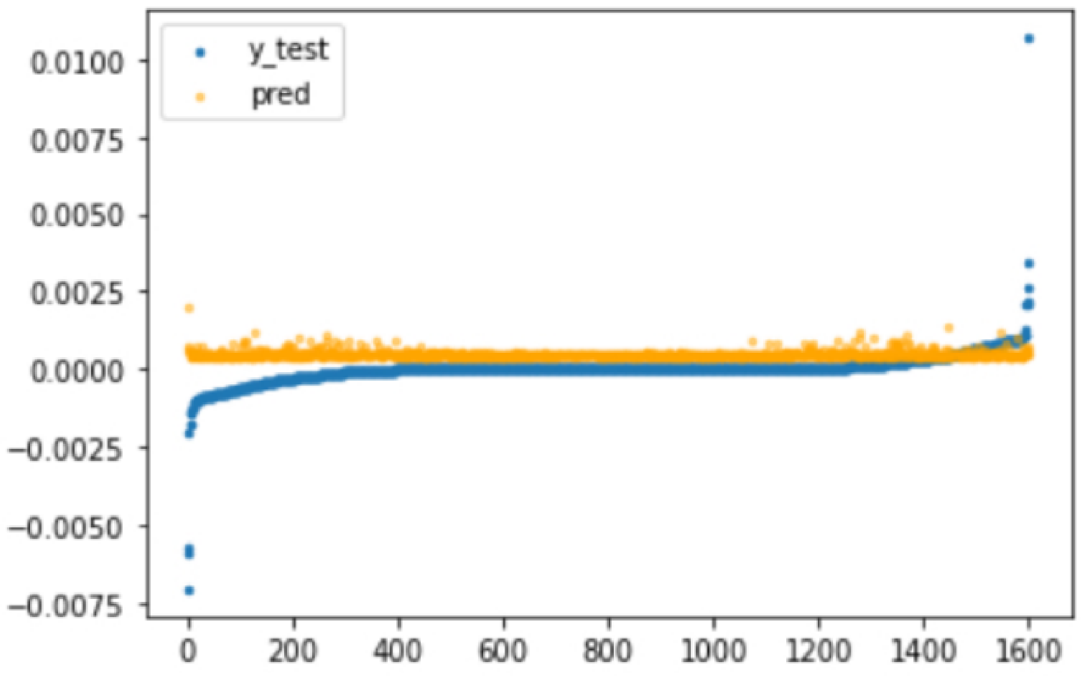
Once we found the appropriate general structure we moved to experimenting with different activations and activation layers, Dropout values and Nominalization layers. We tried many different Dropout values and found that as the input dimensionality gets smaller in the sequential structure the smaller the Dropout value should get. Another important thing we found in our experimentation is that for our situation BatchNominalization does not work as well as LayerNominalization. Once we had a good structure with most of the kinks worked out we started experimenting with different optimizers and activations.

Phi, Theta and Psi Model Structures and results

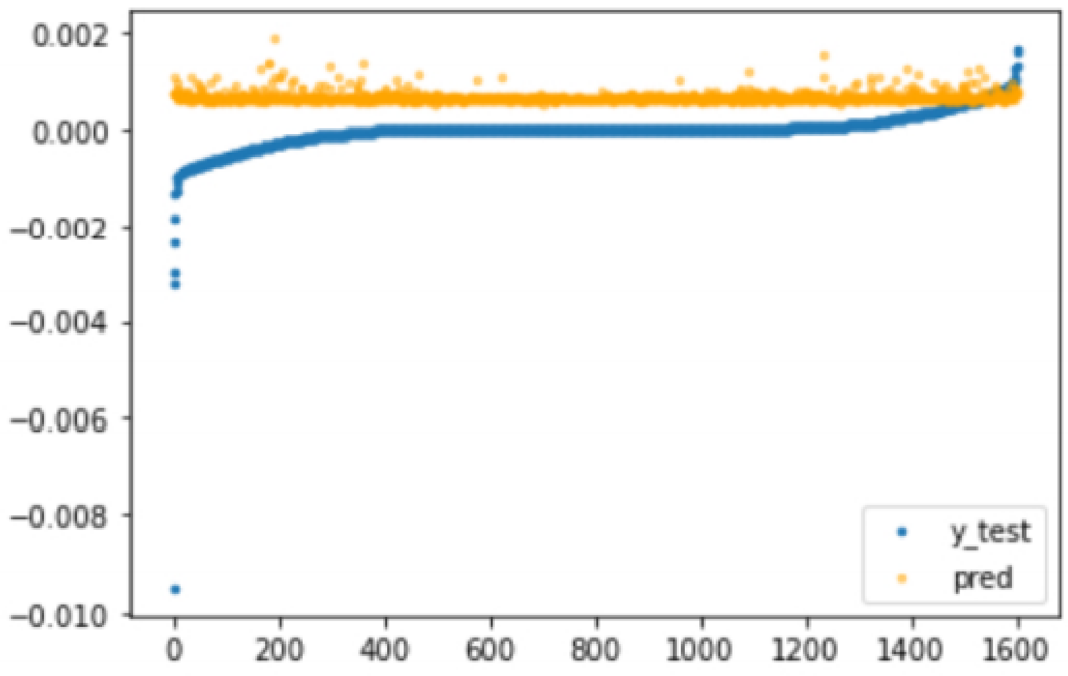
This was our best structure for phi_change, theta_change and psi_change:



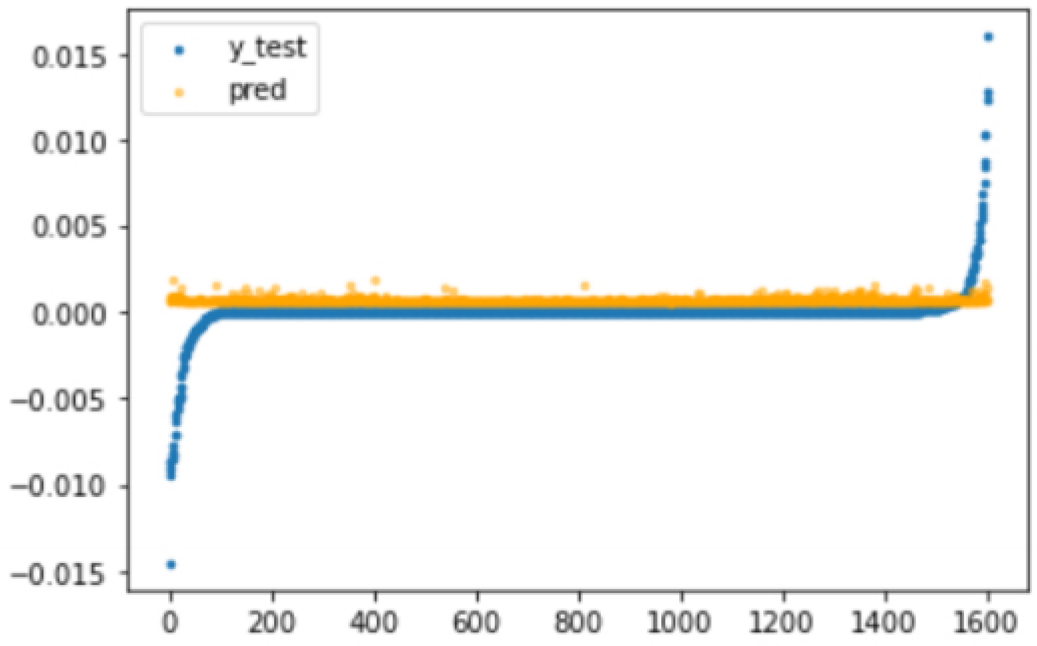
Phi_change actual vs prediction graph for above structure:



Theta_change actual vs prediction graph for above structure:

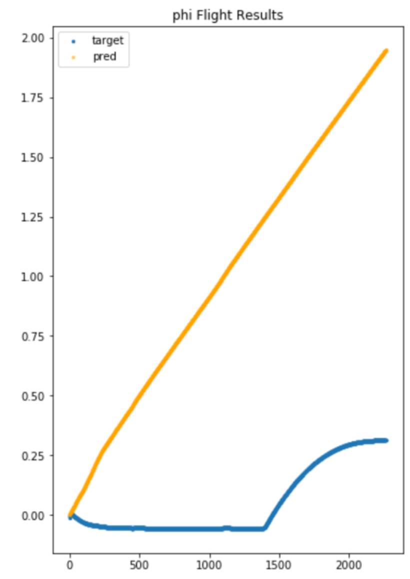


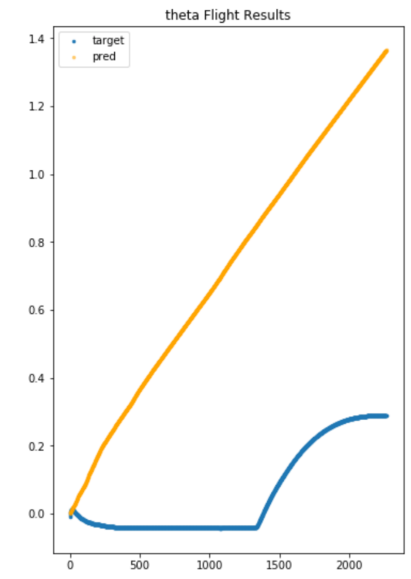
Psi_change actual vs prediction graph for above structure:

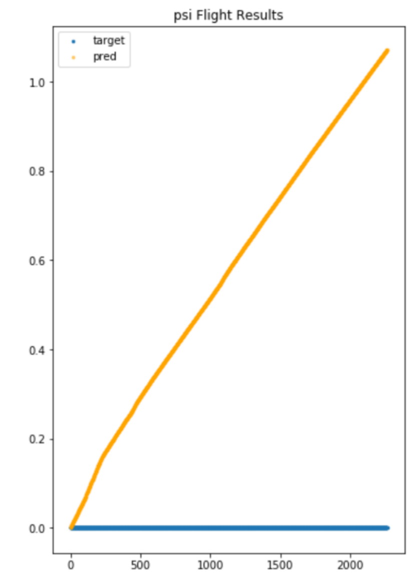


By adding Dropout layers we were able to significantly increase performance. The Dropout layer allowed us to solve our overfitting issues. The Dropout layer does this by randomly dropping nodes. This random dropping of nodes decreases the amount of noise from the training data. This noise is causing the model to not generalize as well. We also added nominalization layers. Nominalization layers allow each layer of the network to do learning more independently. They are used to normalize the output of the previous layer. The layer is added to the sequential model to standardize the inputs or the outputs. These layers can be placed at several points in between the layers of the model. For our problem we found Layer normalization works best instead of Batch normalization. Layer normalization normalizes input across the features instead of normalizing input features across the batch dimension like Batch normalization. Batch normalization struggles when batch size is variable.

The figure below shows the flight path change to predicted flight path, phi/theta/psi:





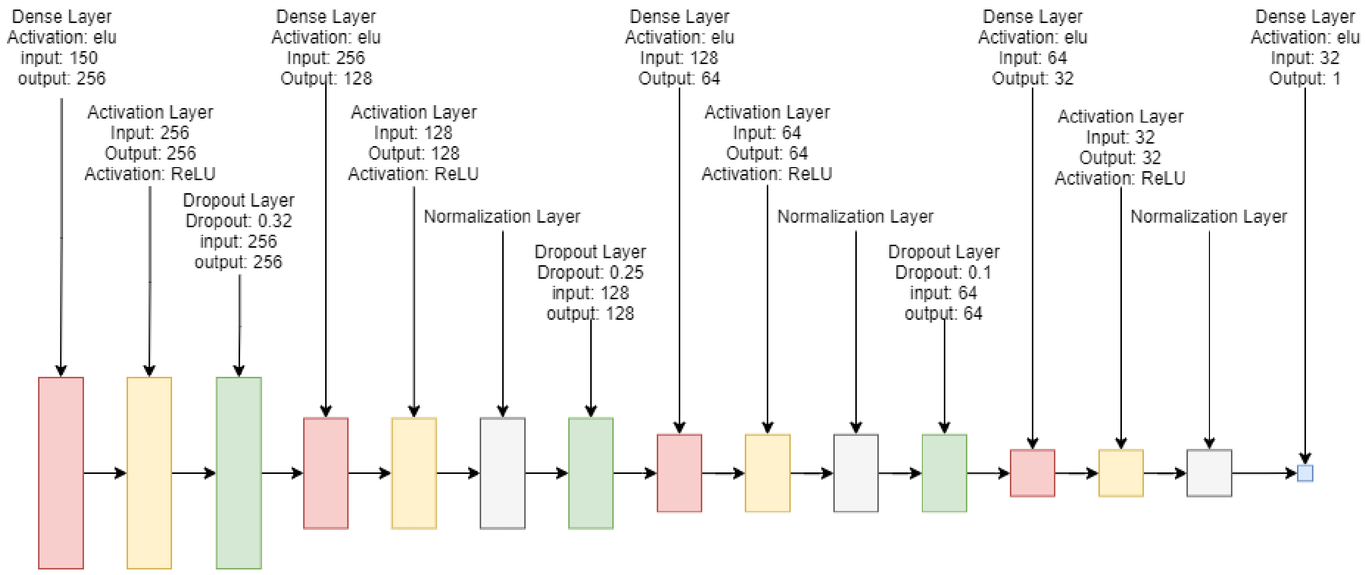


Analysis of phi_change, theta_change and psi_change results

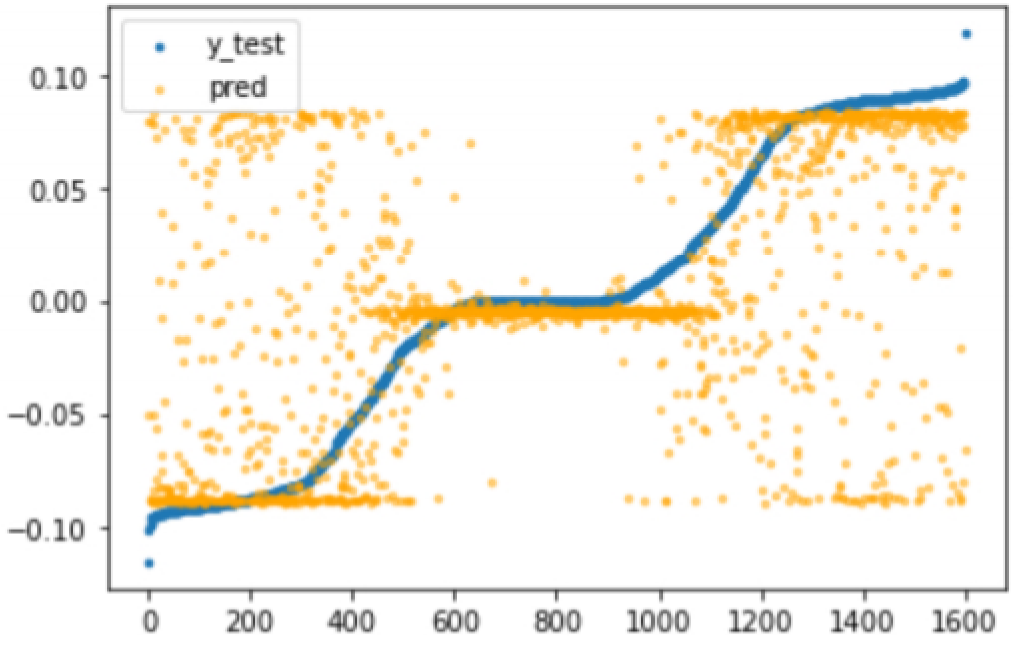
What we realized from these results was the fact that some of the actual values were negative values. This is a problem because we used sigmoid activation. Sigmoid activation function transforms all the input values into a range between 0 and 1.0. This transformation makes it impossible for the DNN structure to predict negative values. Finding a different structure that works for phi_change, theta_change and psi_change and activation functions such as elu and selu would be something we need to do in the future.

X, Y and Z Model Structures and results

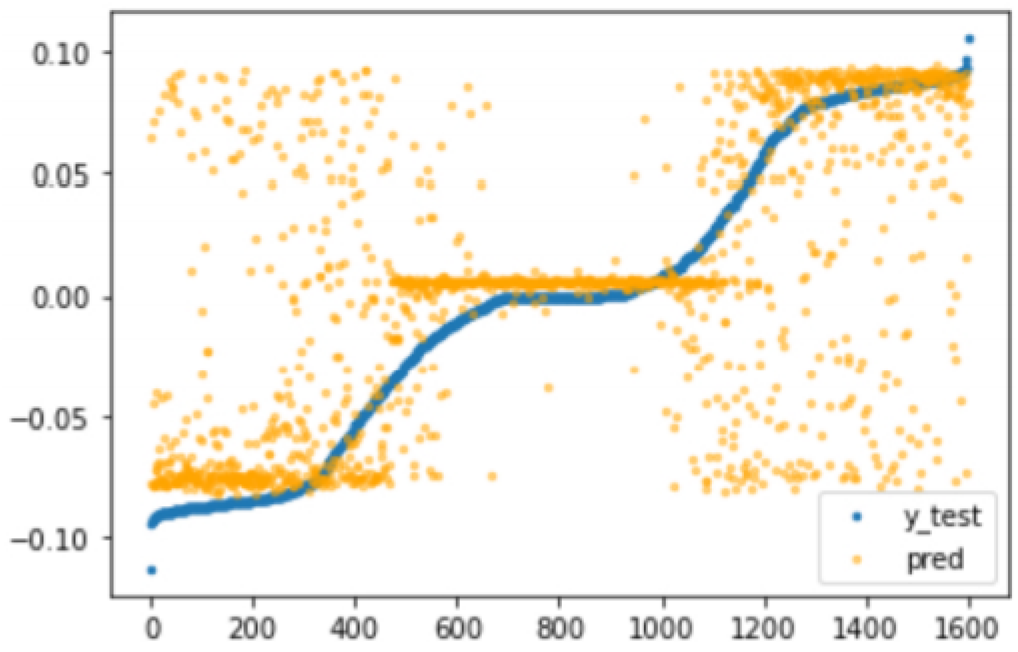
This was our best structure for x_change, y_change and z_change.



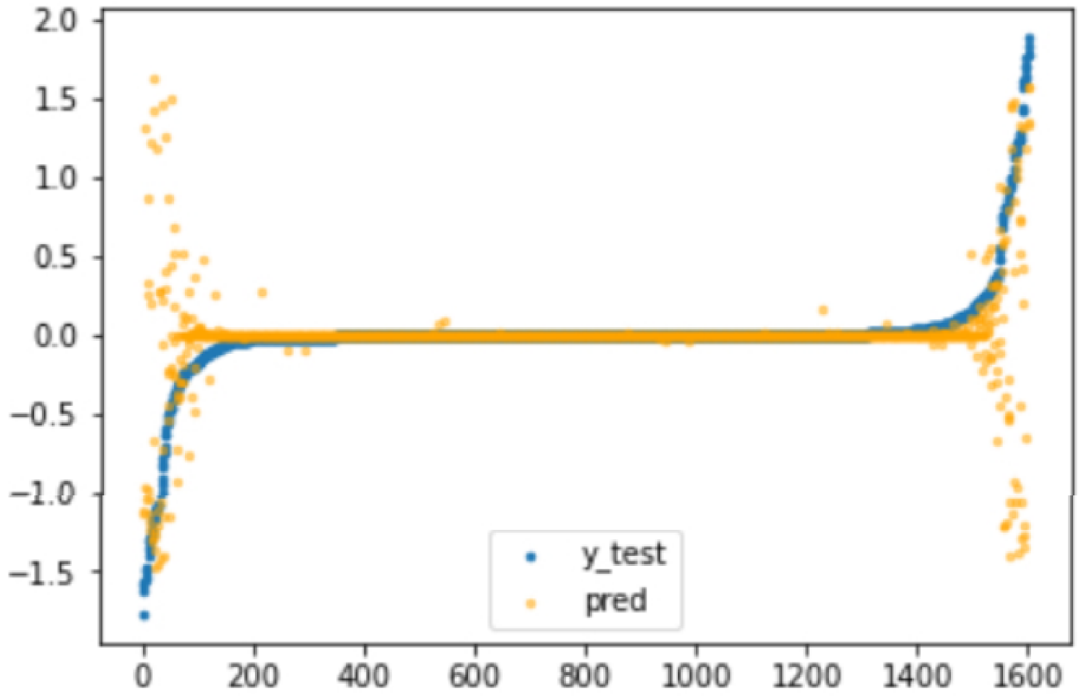
X_change actual vs prediction graph for above structure:



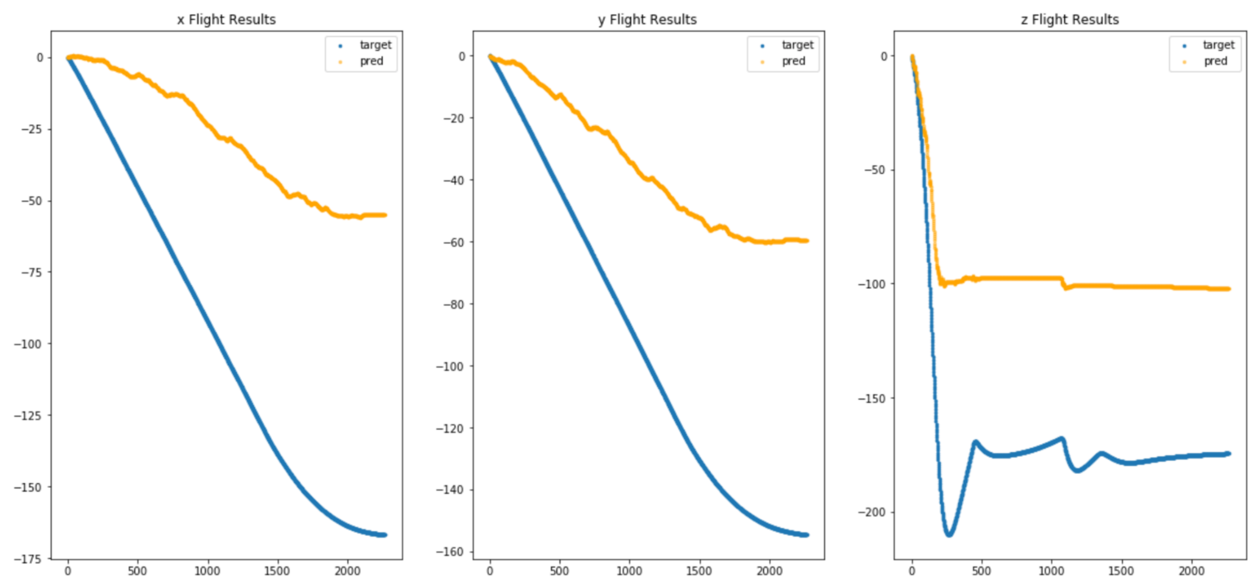
Y_change actual vs prediction graph for above structure:



Z_change actual vs prediction graph for above structure:

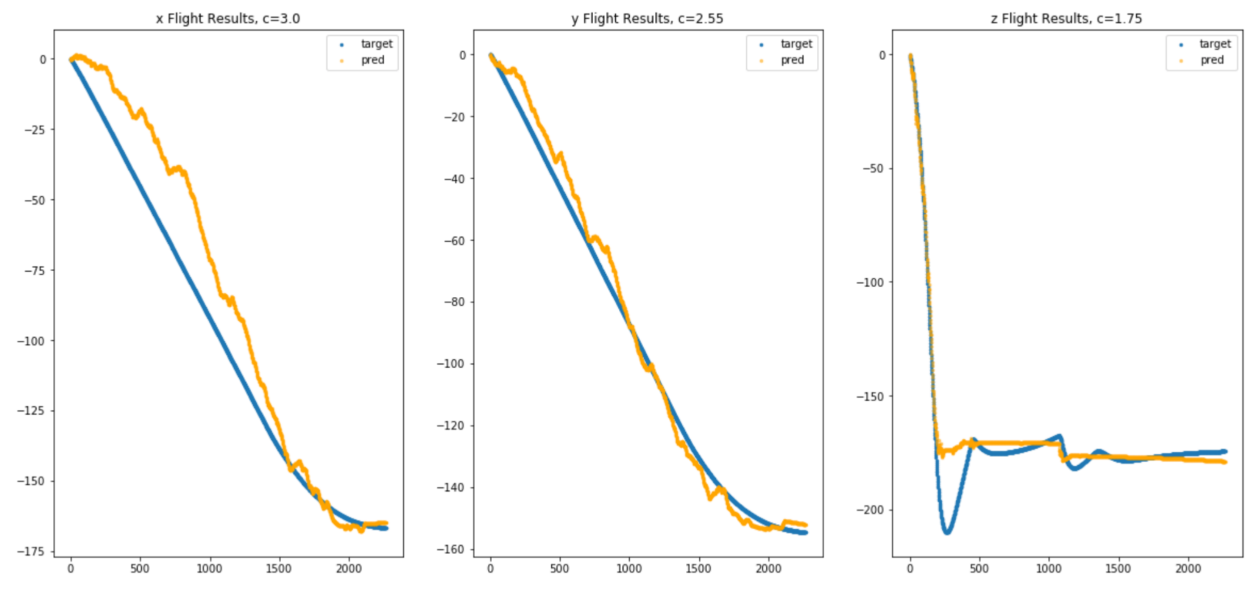


The figure below shows the flight path change to predicted flight path, X/Y/Z:

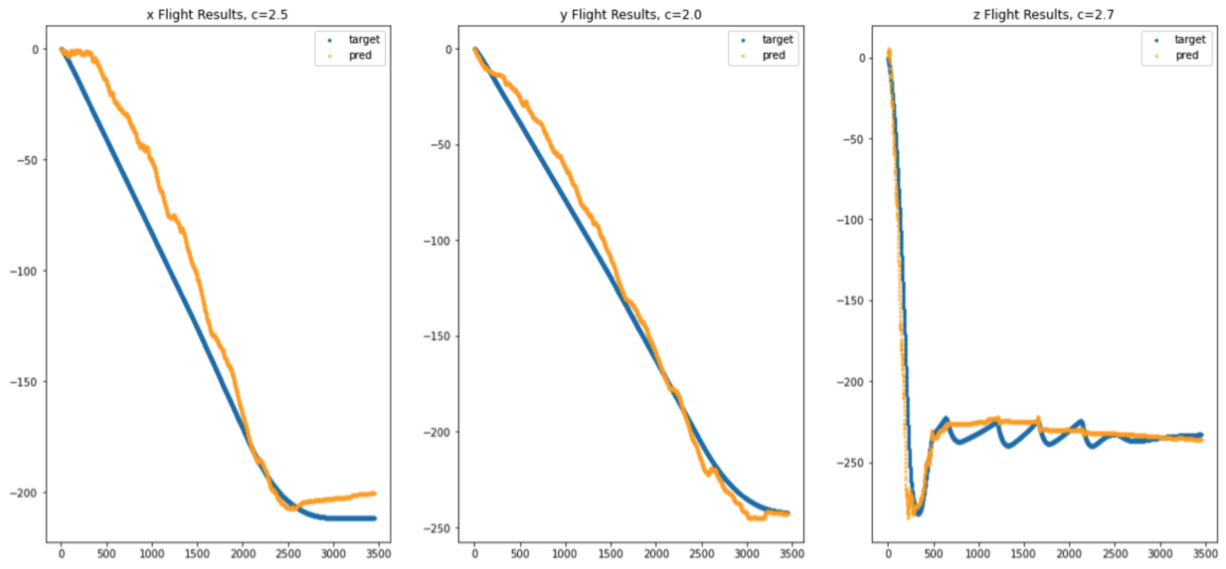


While our predicted flight path did not exactly match the actual flight path the model was able to get the overall shape pretty close.

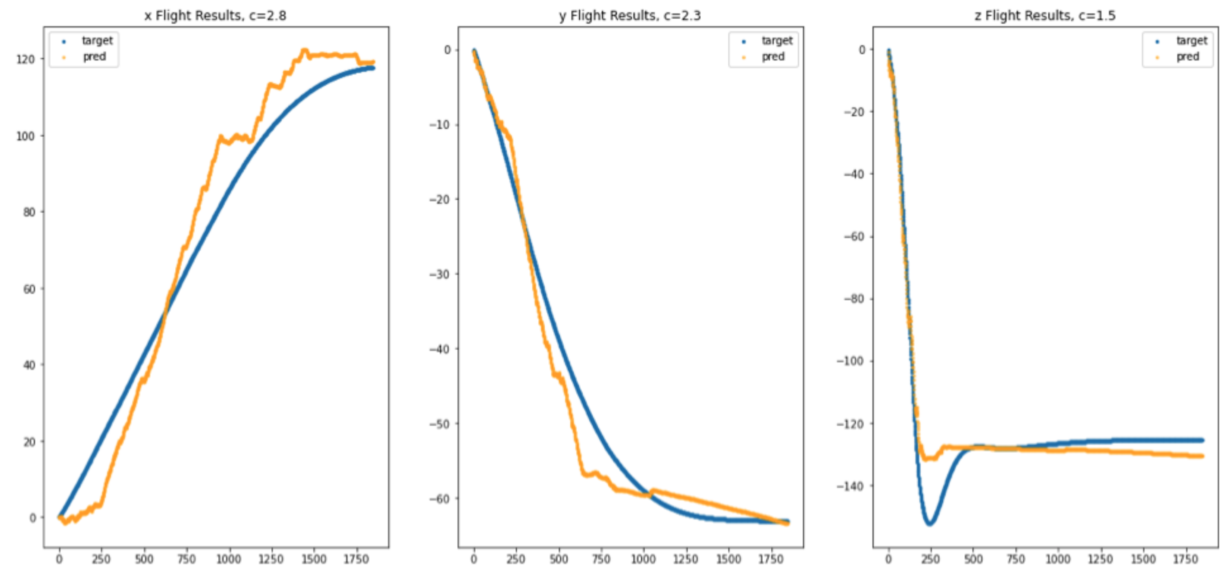
The figure below shows the same flight path but we added constants to the predicted flight path, X/Y/Z: (Constant for x = 3.0, y = 2.55, z = 1.75)



The figure below represents another flight path along with the predicted path with constants: (Constant for x = 2.5, y = 2.0, z = 2.7)



Another example: (Constant for x = 2.8, y = 2.3, z = 1.5)



Adding the constant allows us to get a better understanding of the predictions and where the model was lacking.

Analysis of x_change, y_change and z_change results

What we realized from these results was the fact that though the predictions follow the actual values pretty well the model is still incapable of differentiating the edge case predictions. This can be solved by simulating flights where the quadcopter spends less time stationary and more time changing its positions. This change allows for more data to exist with less repetitive data.

Conclusion

In conclusion, the models we came up with worked relatively well as we got a good RMSEE scores compared to Al-Sharman et al. As a group we've gotten better with deep learning libraries such as Keras and Tensorflow. We were able to perform data processing very efficiently. Preprocess timestamped simulation data into a vector format. Learn how to apply DNN Regression to a Vector of IMU data. Learn Fast R-CNN for object detection. However, while working on the project we had few challenges that we had to overcome. When relying on IMU sensor data to train the model one issue is that it's dependent on the sensor we select. Each sensor has its own error dynamics wherein the noisy output may be directed towards a specific average value or the output has a certain Gaussian noise. This means that, were we to train our model on a specific IMU, then the model might only work on the specific IMU. Another issue that represents itself is there is currently no data published by Al-Sharman et al. nor DLNN code, that can be utilized to help train a model. This produces 2 main challenges: The sensor that we train the model on determines the sensor that we need to utilize when feeding data to the model. There is no data available to recreate Al-Sharman et al.'s model, let alone train our own model. To resolve both issues, we need to somehow create data that will reflect the real state of the UAV, while also providing the current IMU readings of the accelerometer and gyroscope. This data also needs to represent multiple different sensors, with a broader range of error dynamics that can reflect varying IMUs. The data also needs to represent multiple different flights to allow various error dynamics of the IMU to show themselves. To do this we utilize Micklisch et al.'s simulator to generate the data, which utilizes simulated sensors to comprise a IMU with broad error dynamics. The simulator can be run to output a CSV that represents a single flight path which will contain the real state of the UAV in the simulated environment along with the accelerometer and gyroscope readings of the IMU. This data is pulled during the flight every 0.03 seconds. Utilizing this simulator we can produce hundreds of random flight path CSVs which can then be preprocessed into a single CSV to train the model. (This data would be simulated and self-made).

References

- Al-Sharman, Mohammad K., et al. "*Deep-learning-based neural network training for state estimation enhancement: Application to attitude estimation.*" IEEE Transactions on Instrumentation and Measurement 69.1 (2019): 24-34.
- Aznar, Fidel, Mar Pujol, and Ramón Rizo. "*Visual navigation for uav with map references using convnets.*" Conference of the Spanish Association for Artificial Intelligence. Springer, Cham, 2016.
- Bableshwar, Anusha Anand. *A Recurrent Neural Network Based Patch Recommender for Linux Kernel Bugs* . 2019. Ann Arbor, MI.
- Carrio, Adrian, et al. "*A review of deep learning methods and applications for unmanned aerial vehicles.*" Journal of Sensors 2017 (2017).

- Caterini, Anthony L., and Dong Eui. Chang. *Deep Neural Networks in a Mathematical Framework*. 1st ed. 2018. Cham: Springer International Publishing, 2018. Web.
- Cheon, Kangbeom, et al. "*On replacing PID controller with deep learning controller for DC motor system.*" Journal of Automation and Control Engineering Vol 3.6 (2015).
- Deshpande, Mayuri. *Object Tracking Using Convolutional and Recurrent Neural Network* . 2018. Ann Arbor, MI.
- Girshick, Ross. "Fast r-cnn." Proceedings of the IEEE international conference on computer vision. 2015.
- Goyal, Palash., Sumit. Pandey, and Karan. Jain. *Deep Learning for Natural Language Processing Creating Neural Networks with Python* . 1st ed. 2018. Berkeley, CA: Apress, 2018. Web.
- Ibarra-Bonilla, Mariana N., P. Jorge Escamilla-Ambrosio, and Juan Manuel Ramirez-Cortes. "*Attitude estimation using a Neuro-Fuzzy tuning based adaptive Kalman filter.*" Journal of Intelligent & Fuzzy Systems 29.2 (2015): 479-488.
- Jaradat, Mohammad Abdel Kareem, and Mamoun F. Abdel-Hafez. "*Non-linear autoregressive delay-dependent INS/GPS navigation system using neural networks.*" IEEE sensors journal 17.4 (2016): 1105-1115.
- Jin, Ren, et al. "*Drone detection and pose estimation using relational graph networks.*" Sensors 19.6 (2019): 1479.
- Kemper, E . *Quad-rotor flight path energy optimization*. Doctoral Dissertation, Northern Arizona University.
- Micklisch, Christian, Nathan Hilliard, and Hala ElAarag. "*Modeling and simulation of autonomous quadrotor systems and their onboard sensors.*" SIMULATION (2020): 0037549720974753.
- Nirmal, K., et al. "Noise modeling and analysis of an IMU-based attitude sensor: improvement of performance by filtering and sensor fusion." *Advances in Optical and Mechanical Technologies for Telescopes and Instrumentation II*. Vol. 9912. International Society for Optics and Photonics, 2016.
- Yang, Hyunsoo, et al. "*Multi-rotor drone tutorial: systems, mechanics, control and state estimation.*" Intelligent Service Robotics 10.2 (2017): 79-93.