Network-based Inertial Navigation

S. G. Andersen, M. Bach, A. G. Golles, A. A. Sousa-Poza & K. Tavanxhi

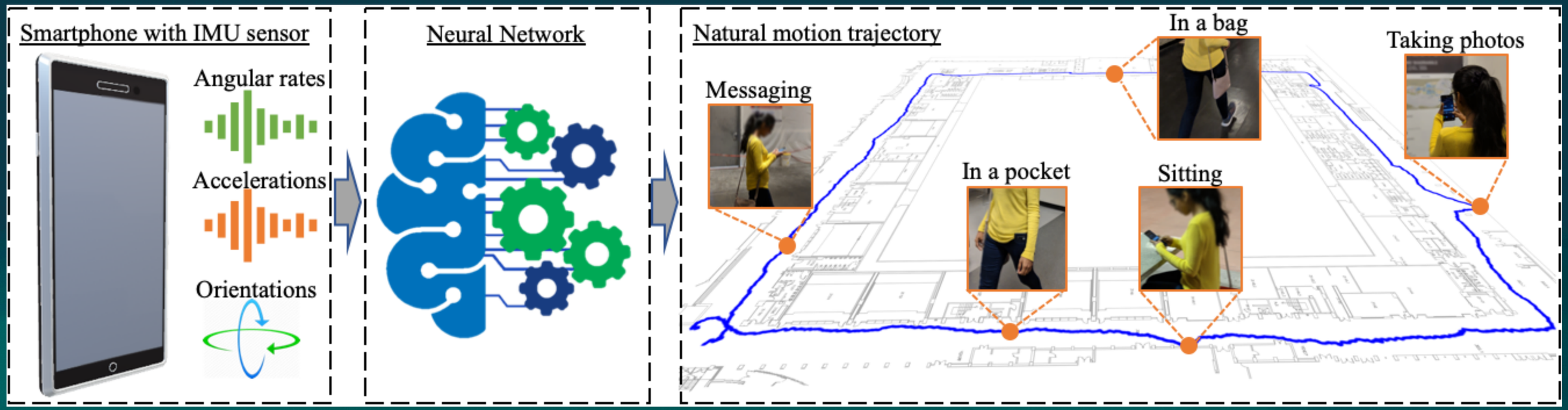
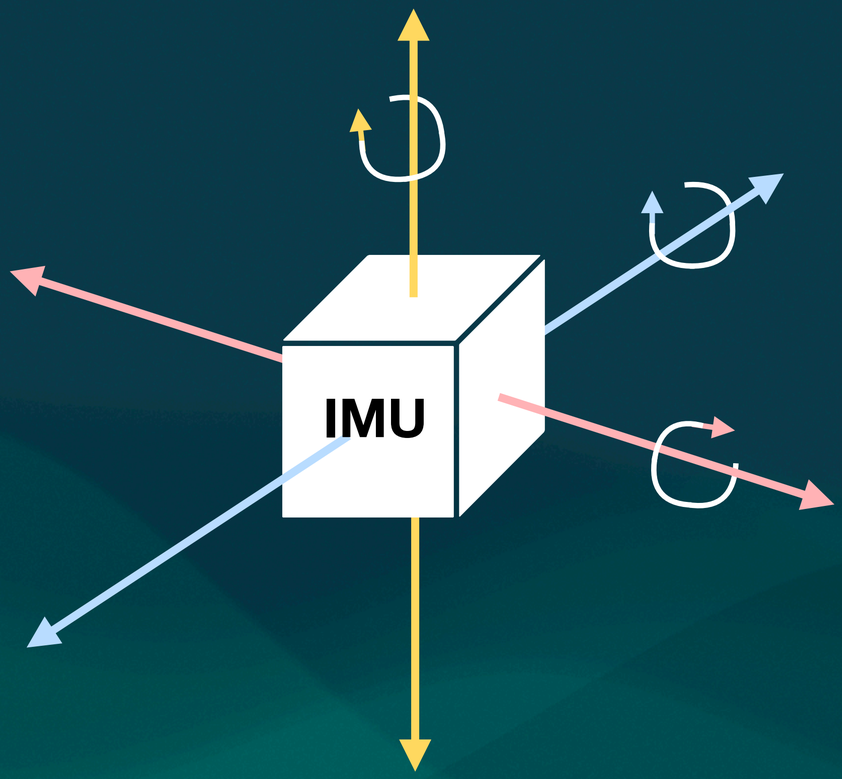


Illustration from https://ronin.cs.sfu.ca/

Agenda

- Data
- Typical IMU processing
- End-to-end architecture
- Feature extension (clustering)
- Gyroscope integration
  - Dynamic filter combination
- End-to-end network
- Neural accelerometer integration
- Conclusion



The Data

Raw → **Synced**

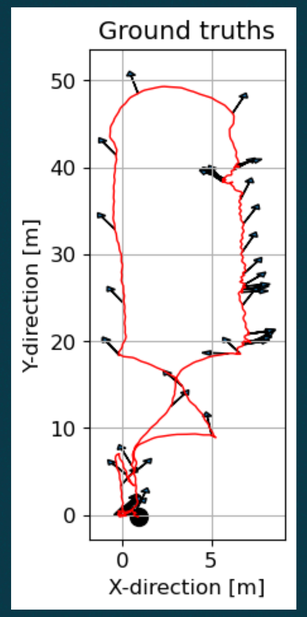
12 Features

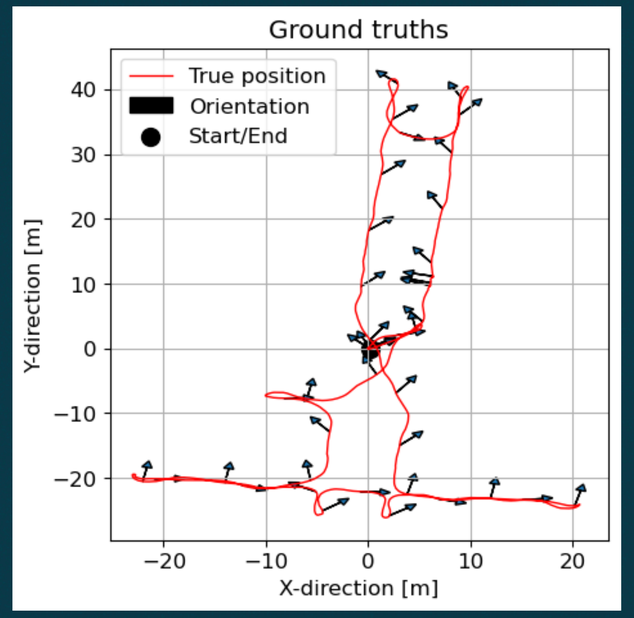
- Accelerometer
- Magnetometer
- Gyroscope
- RV

Pose

2 Ground truths

- True position
- True orientation





100 recorded and synced runs, with 50.000 - 250.000 data points at 200Hz.
Plot displays true position and orientation for sample runs.

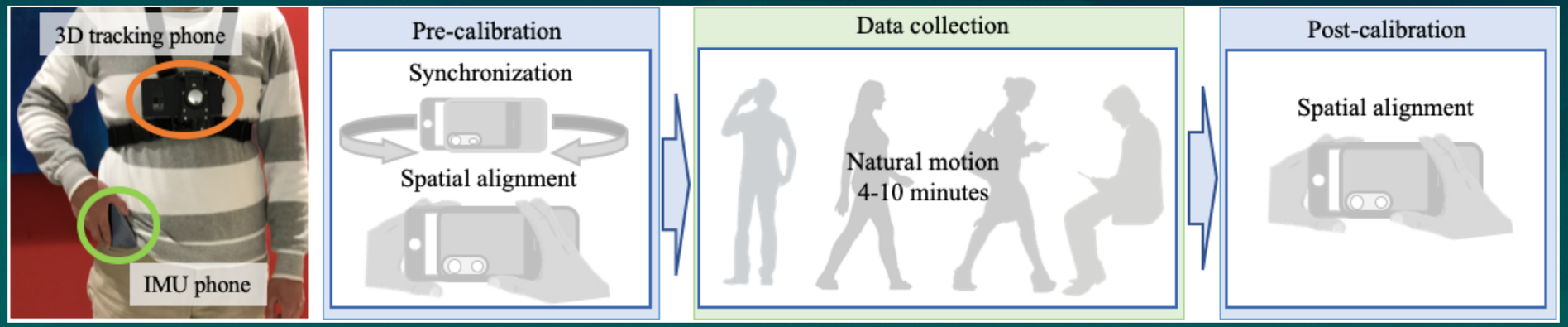
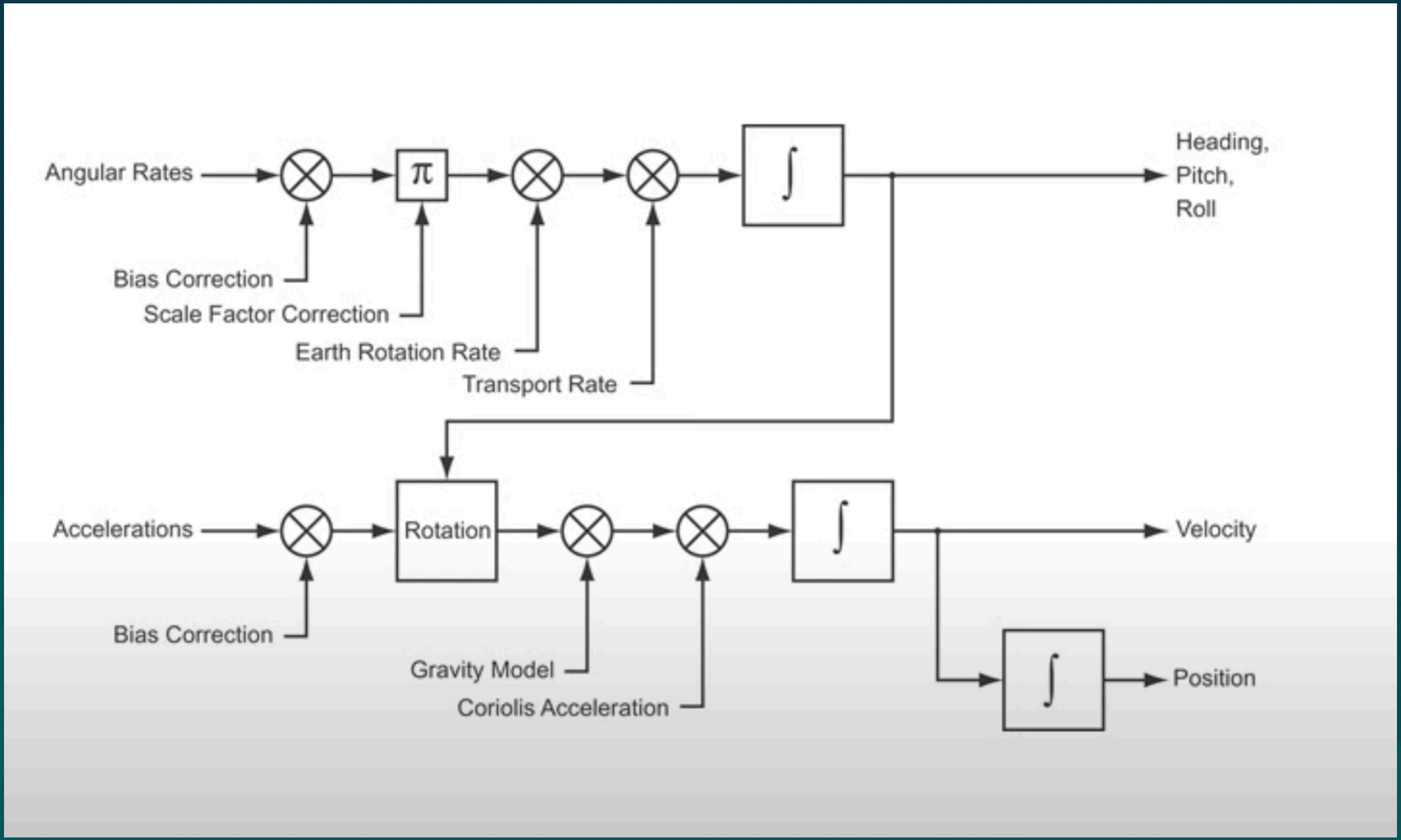
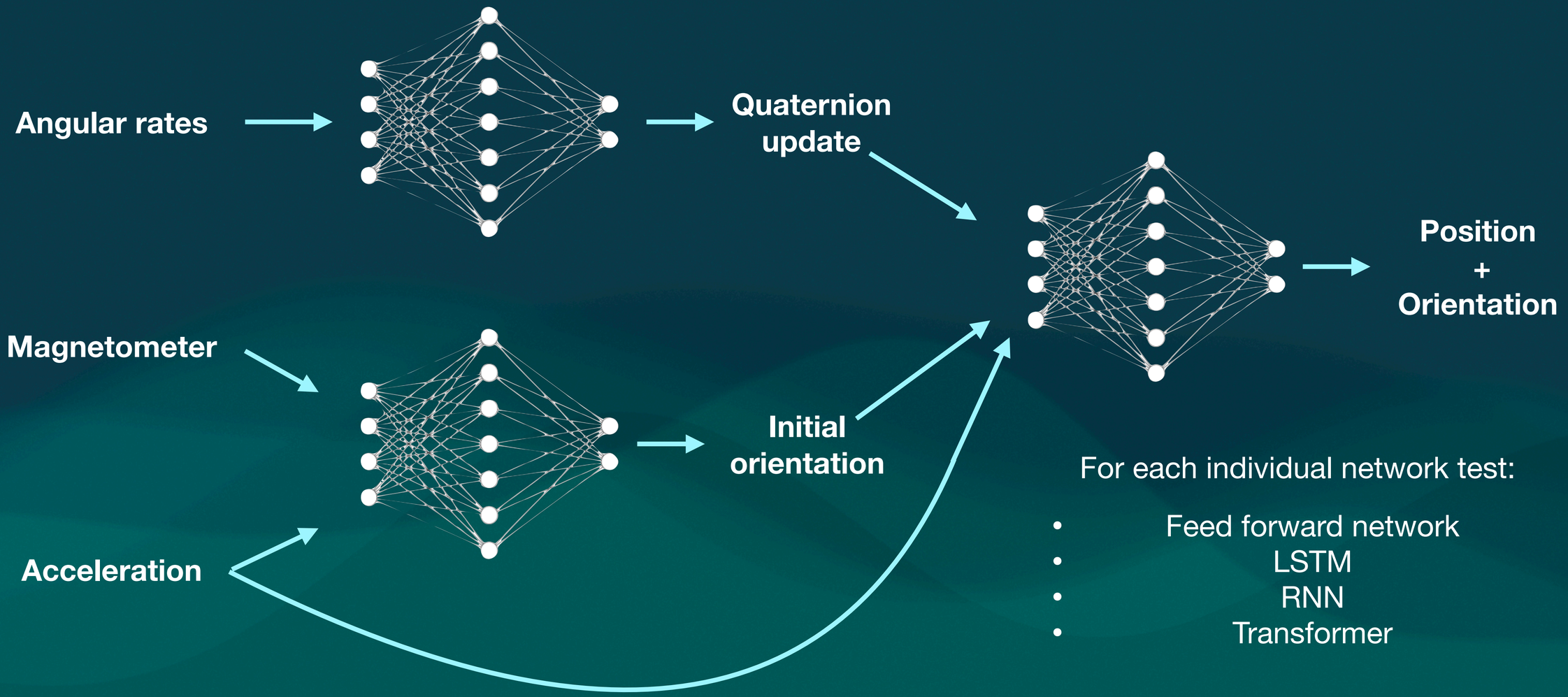


Illustration from https://ronin.cs.sfu.ca/

Naïve IMU processing (double integration)



Network following analytical path



For each individual network test:

- Feed forward network
- LSTM
- RNN
- Transformer

Inertial Navigation Neural Network

Pros and cons

	Analytical approach	Neural Network
Speed	Fast	Maybe slow
Testing requirements	Reliable without much testing	Need vigorous testing to guarantee reliability
Drift	Quadratic positonal drift caused by linear velocity	Increased stability. Since we can identify stand-still!
Perfomance given noisy data	Poor	Potentially stable

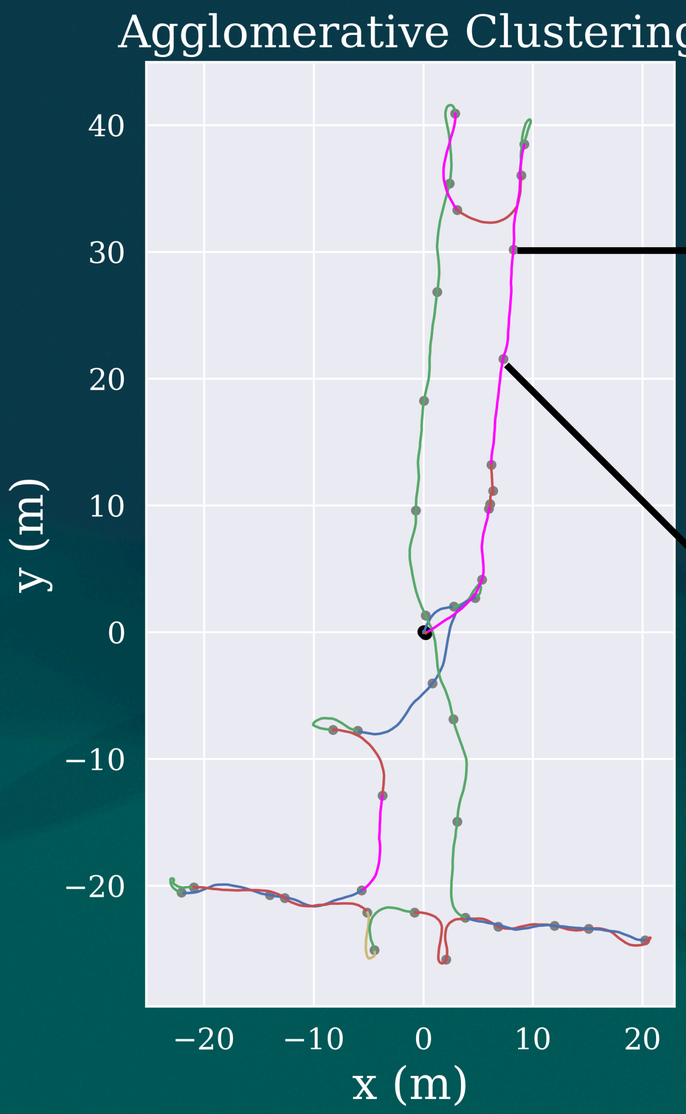
Feature Extension

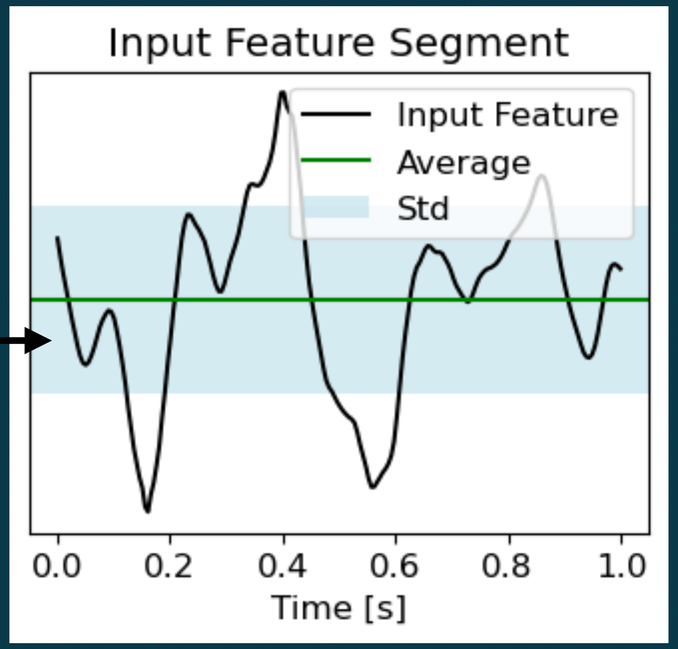
Clustering

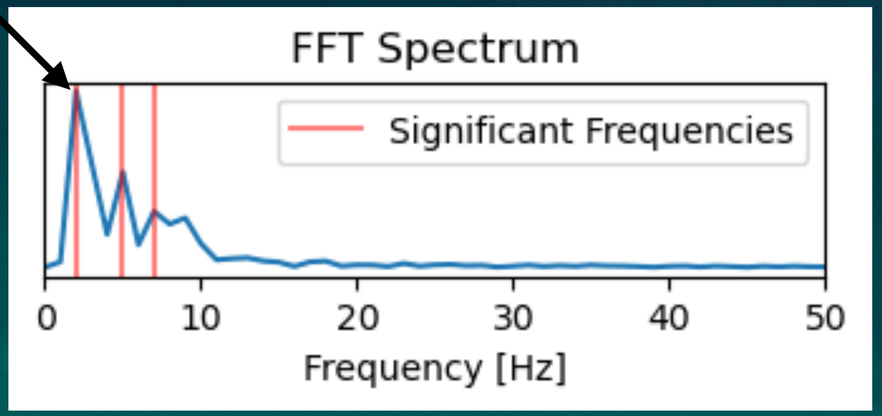
Idea: Extend features with clustering labels applied to processed segments of data.

What segments have similar characteristics or equivalently;

"Where are the patterns in the data?"



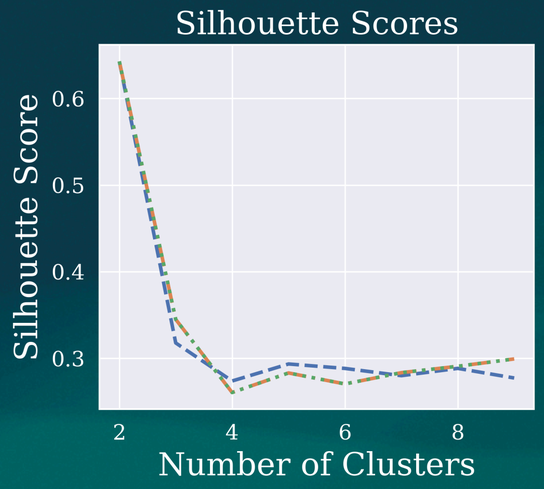


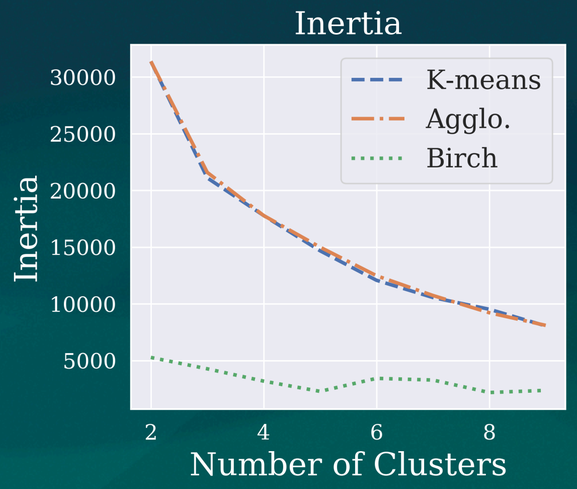


Feature Combinations

Clustering

Input Features	Characteristics	Clustering methods
• Accelerations • Orientations • Angular rates	• Average • Standard Deviation • Significant Frequencies	• Kmeans labels • Agglomerative labels • Birch labels • Consensus



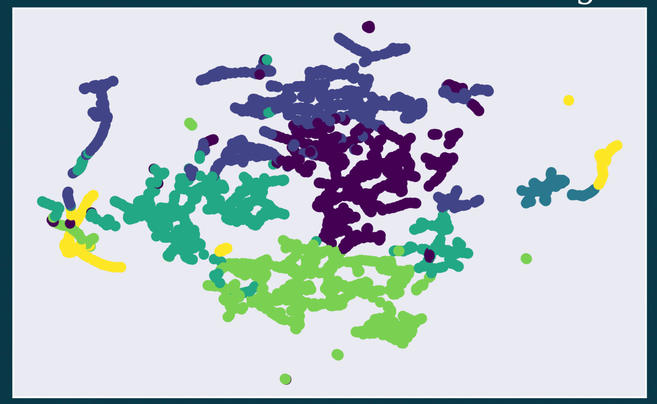


Hyperparameters used

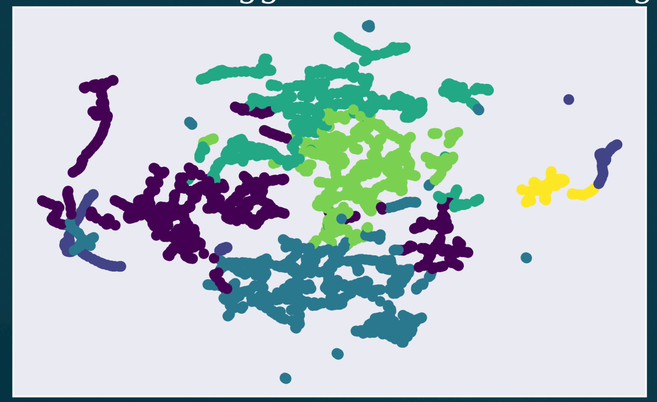
- Number of clusters determined with silhouette score and set to 9
- Significant peaks set to 3
- Two segmentation lengths used
  - Short 0.5s: Captures micromovement like steps and turns
  - Long 5s: Captures macromovement like running or talking on the phone

Extending each feature component with factor 11

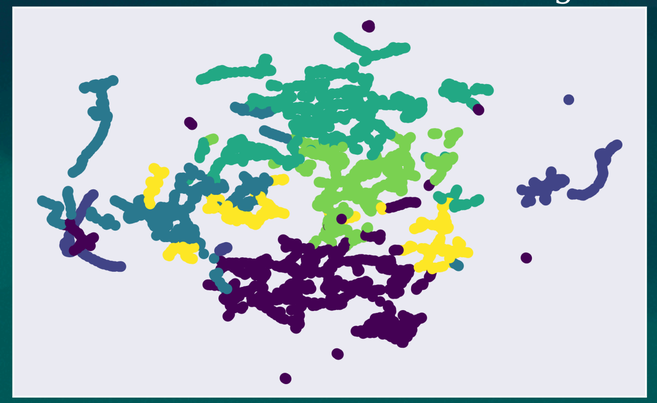
UMAP with Kmeans Clustering



UMAP with Agglomerative Clustering

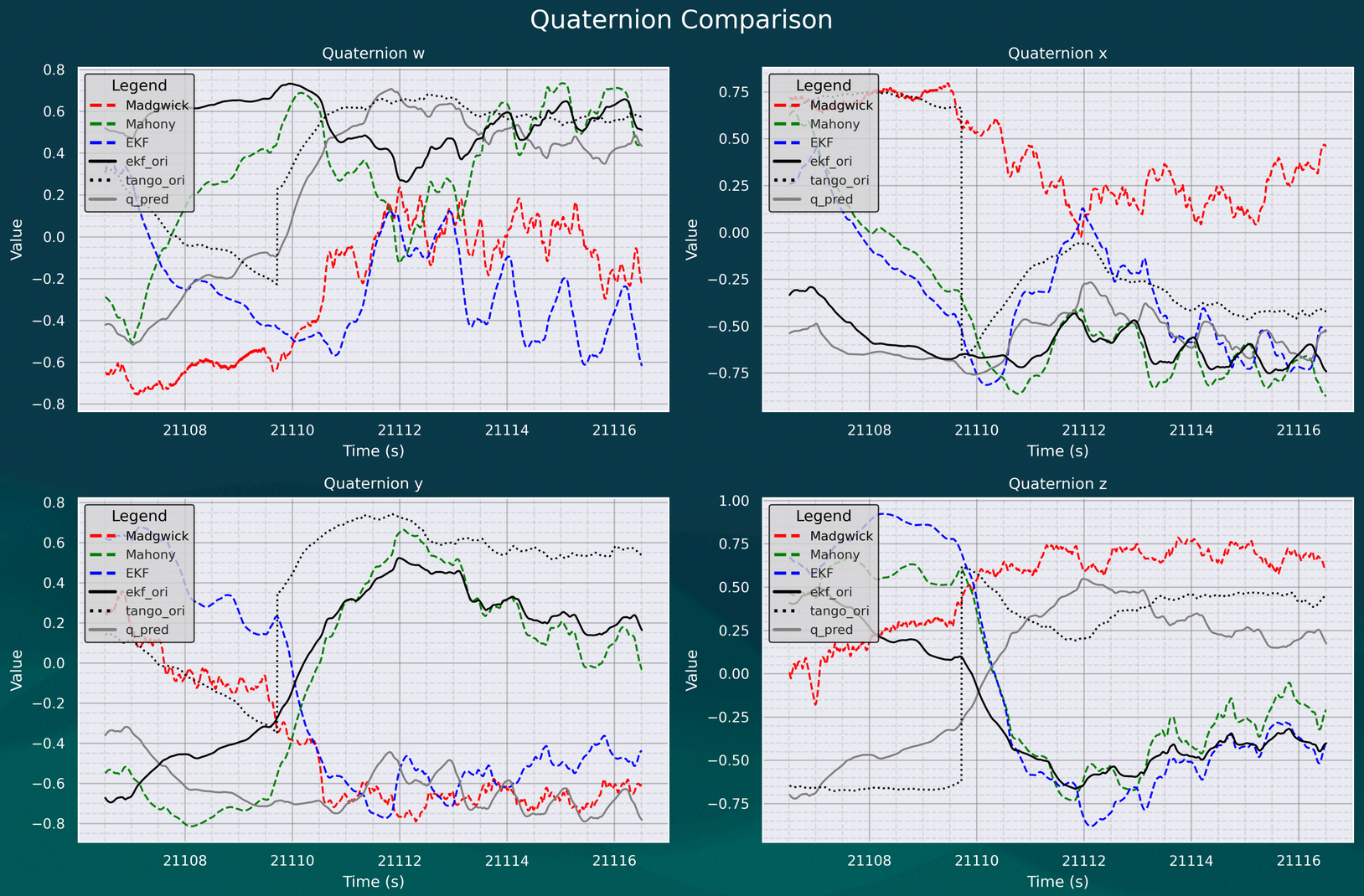


UMAP with Birch Clustering



Filtering

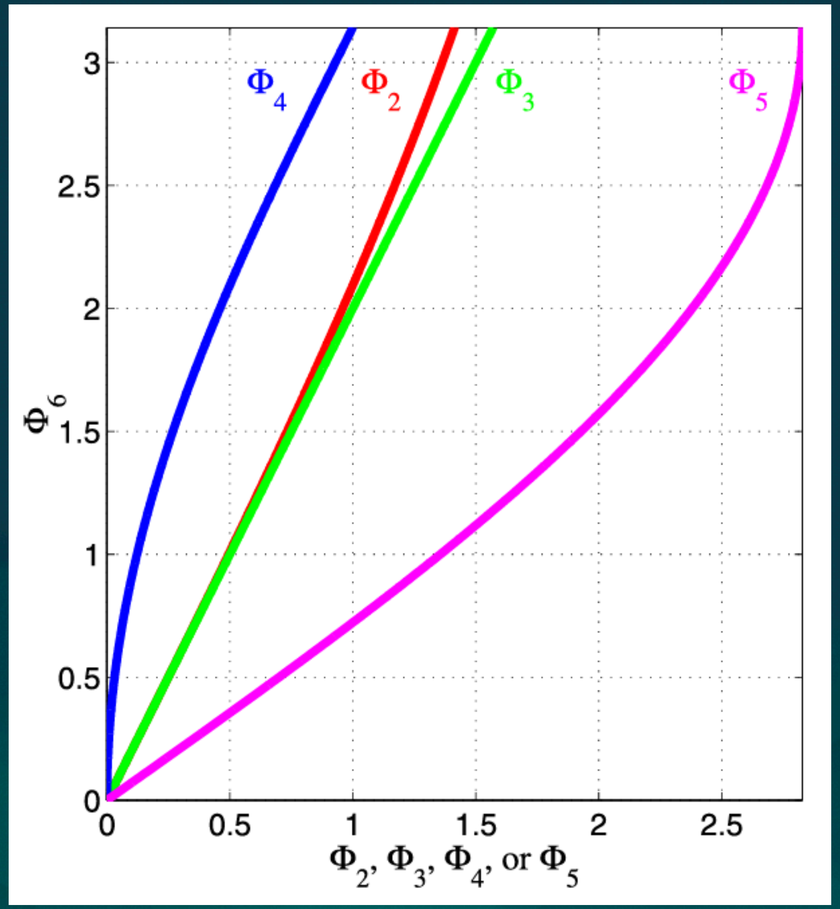
- Problems with integration
- A multitude of analytical filters we applied to the gyroscopic data.
- None yielded a clean fit with the ground truth.



Loss Function for Quaternions

- Quaternions have symmetric properties

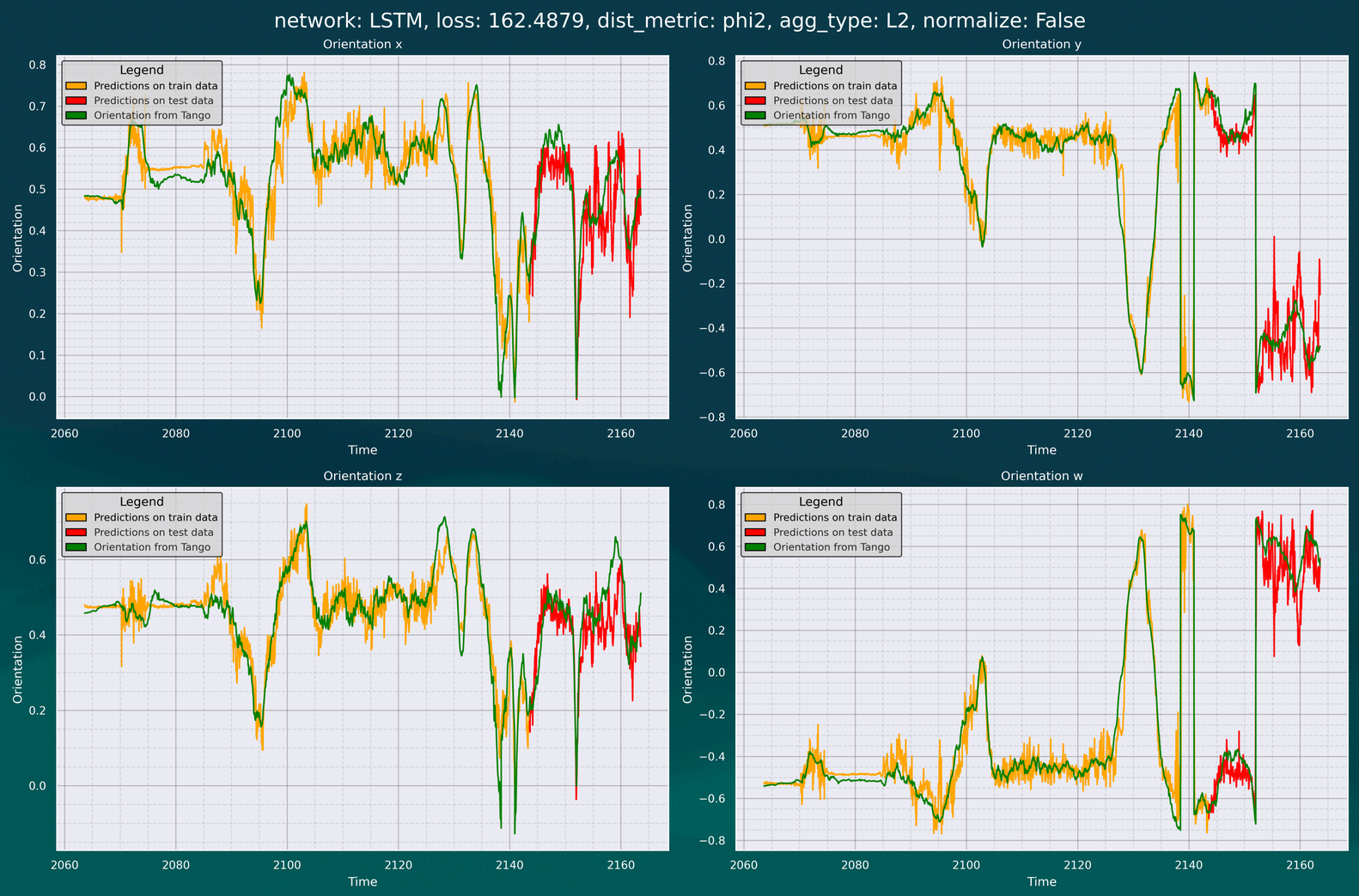
-> New loss function



Dynamic Filter Combination (LSTM)

Quaternion estimation

- Poor performance of individual filters led us to implement dynamic filter-weighting.
- Filters are combined with an LSTM model



Dynamic Filter Combination (comp.)

Quaternion estimation

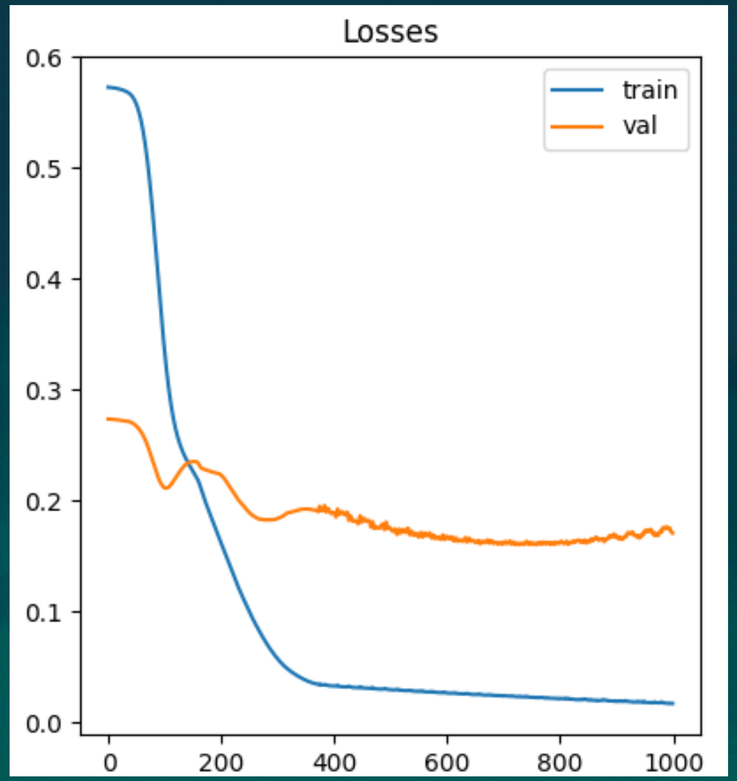
	NN-based models			Analytical approaches			
Model	LSTM	GRU	RNN	Integration	EKF	Mahoney	Madwick
ATE							
RTE							
Runtime							

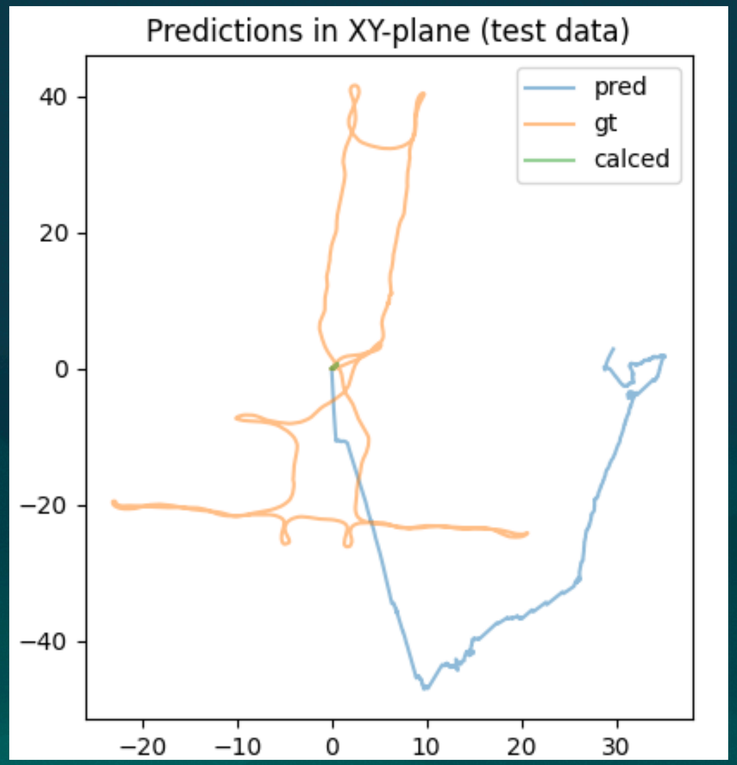
- Poor performance of individual filters led us to implement dynamic filter-weighting.
- Filters are combined with an LSTM model

Position estimation using just IMU data

EKF for quaternion estimation with a Linear network used for position estimation

Despite loss-convergence trace error remains great.

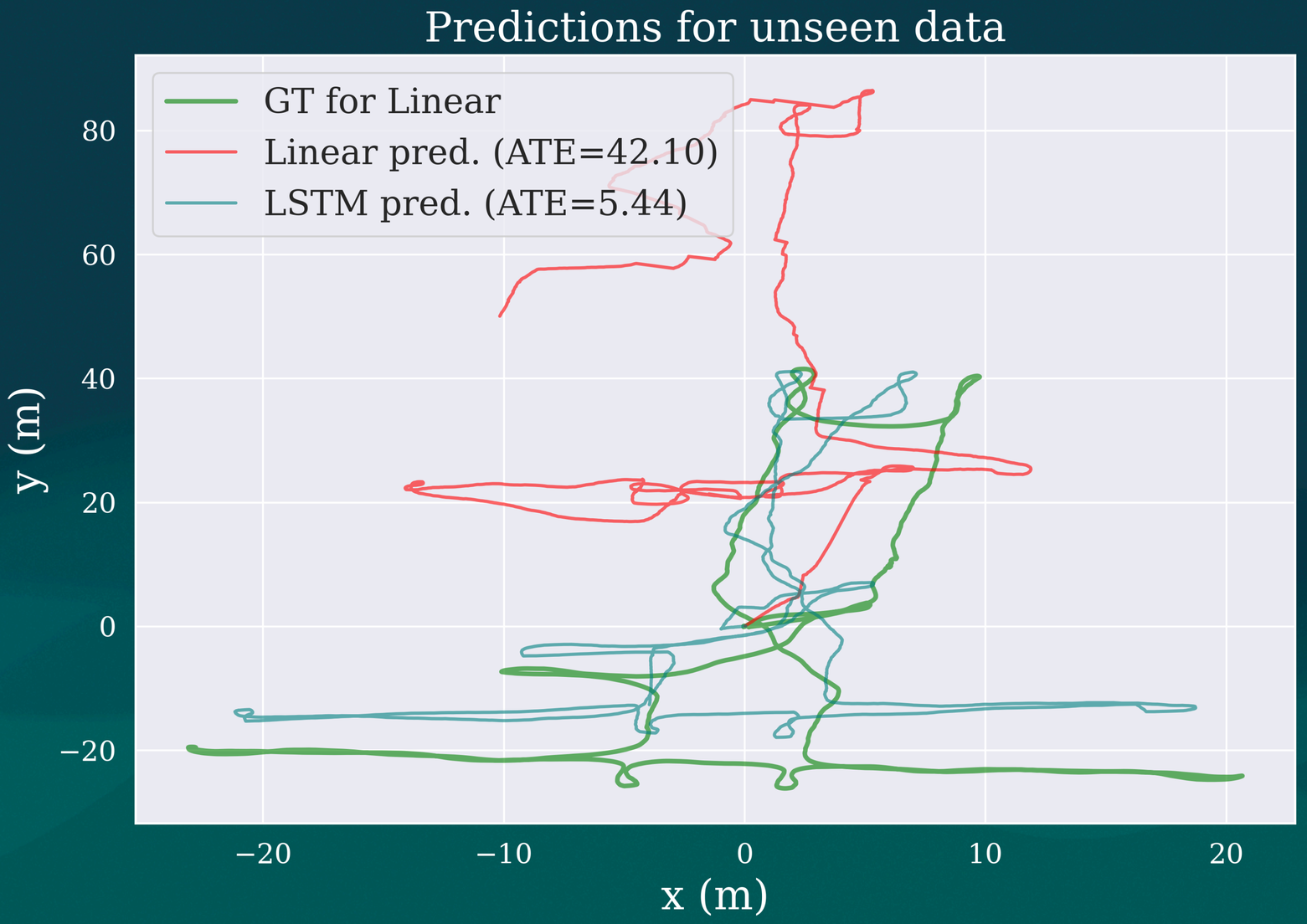




Position estimation using IMU + orientations

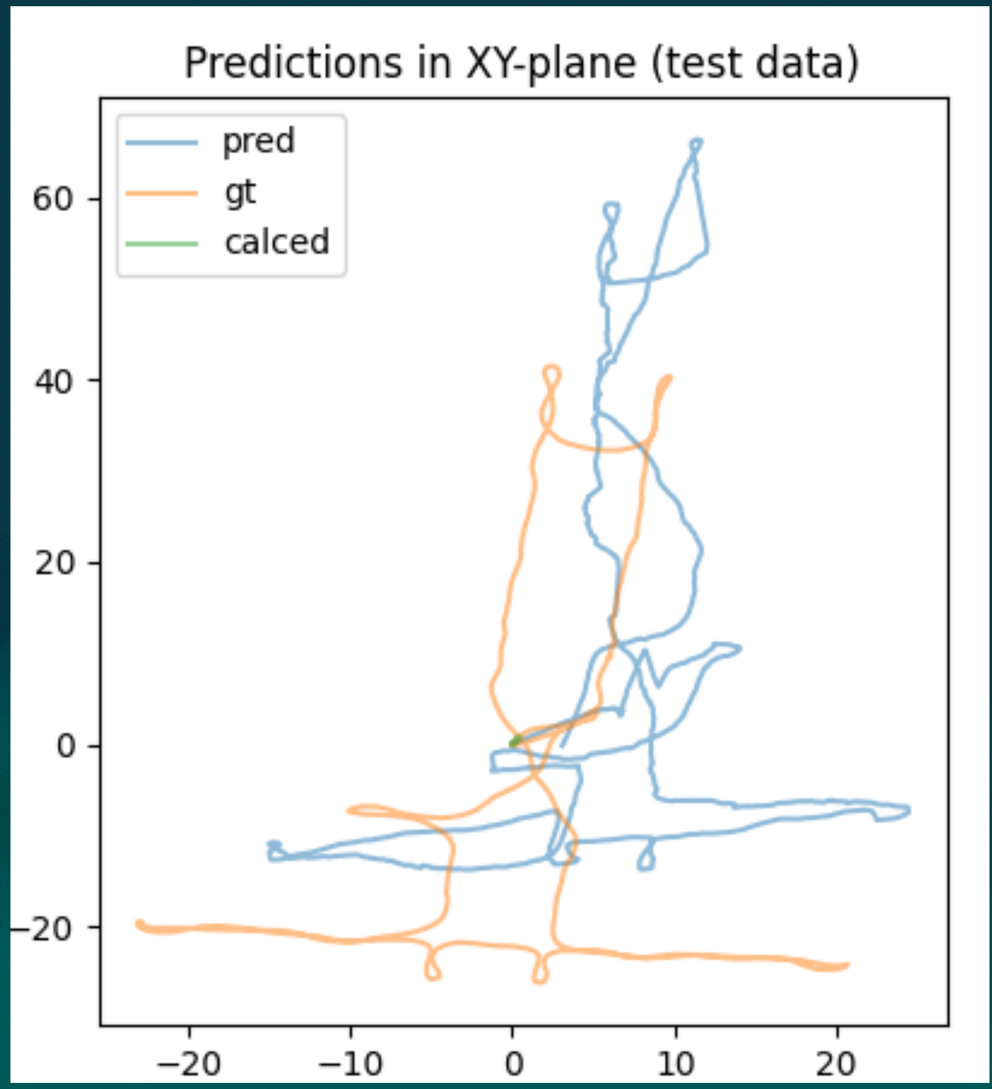
- Position prediction in 'easy mode'
- Architectures: Linear, RNN, GRU, LSTM
- 4.5 million training data points from 75 training sets
- Sequencing data
- Two approaches to predicting

Linear network vs LSTM

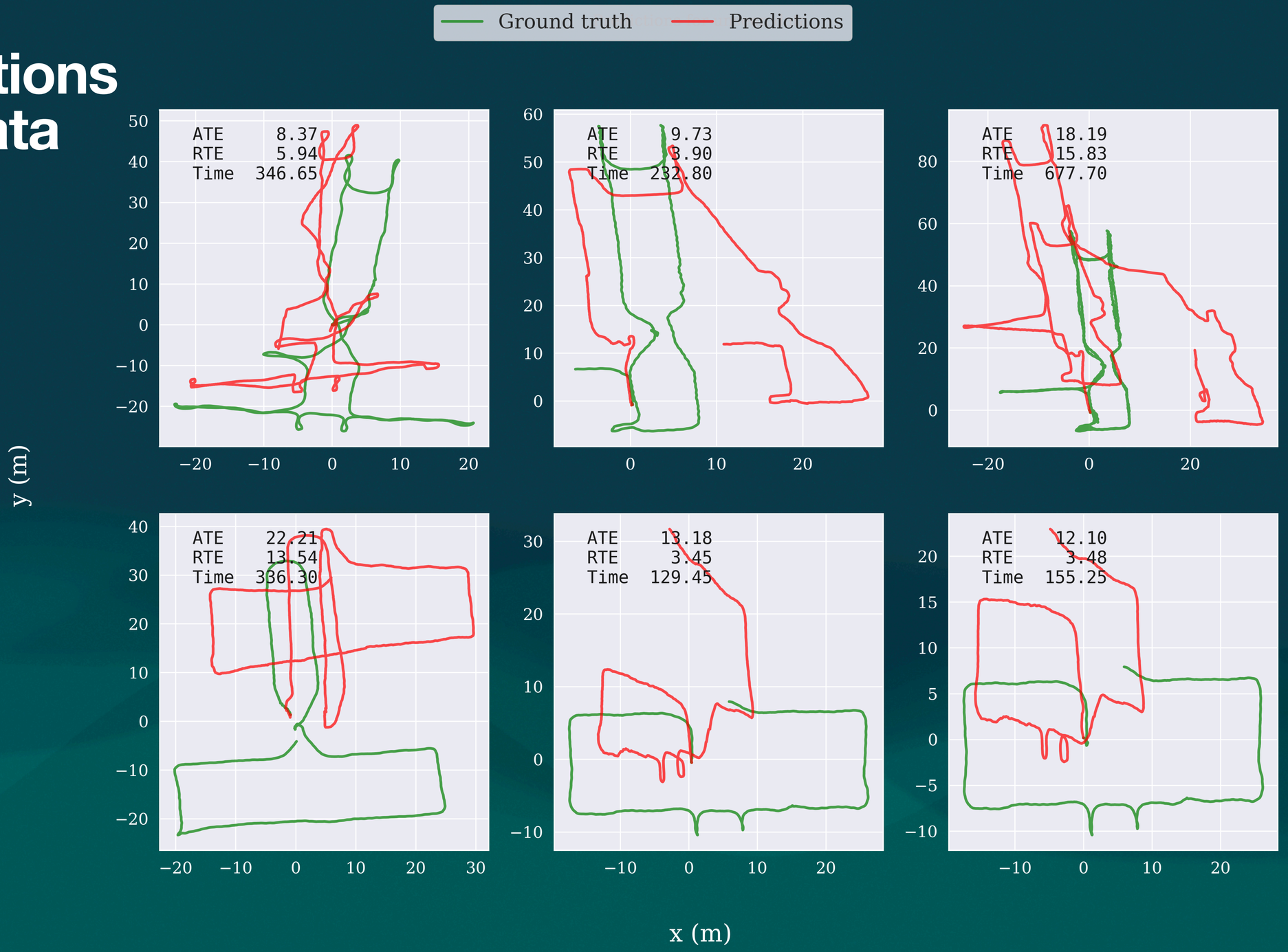


Best run of the linear network

Every once in a blue moon; Anton’s linear network did well.



LSTM predictions on various data sets



LSTM, GRU and RNN performance

	Our models			Benchmark models				
Model	LSTM	GRU	RNN	NDR	PDR	RIDI	IONET	RoNIN LSTM
ATE (m)	32 ± 17	31 ± 15		458.06	27.67	15.66	32.03	5.32
RTE (m)	26 ± 14	25 ± 13		117.06	23.17	18.91	26.93	3.58
Runtime	3m 20s (NVIDIA 940mx 2GB)	5m 10s (NVIDIA 940mx 2GB)	(NVIDIA 940mx 2GB)	?	?	?	?	12 h (NVIDIA 1080Ti 12GB)

- The results of each of our models are calculated as the average performance on 20 recordings
- Information on the benchmark models can be found at https://arxiv.org/abs/1905.12853

Difficulties

- The complexity of the problem necessitated long training times – thus hindering fast exploration.
- Models must be trained on much data (>600,000) in order to learn data structure even roughly
- Other shit

Learnings

- Working with 'Big Data'
- Working with sequential models
- Working with noisy data
- Time series analysis

Appendix

- Analytic quaternion update
- Clustering details
- Effect of supplementing with clustering data
- Filtering method description
- Model descriptions

Analytical Gyroscope Integration

```
def Omega(w=np.array([1,1,2]).reshape(3,1)):
    w = is_column_vector(w)
    w_cross = np.array([[0, -w[2,0], w[1,0]],
                        [w[2,0], 0, -w[0,0]],
                        [-w[1,0], w[0,0], 0]])
    top = np.hstack((-w_cross, w))
    bottom = np.hstack((-w.T, np.zeros((1,1))))
    return np.vstack((top, bottom))

def u_b(w=np.array([1,1,2]).reshape(3,1), dt=.1):
    return  w*dt

# matrix norm of a 4x4 matrix
def matrix_norm(M):
    return np.sqrt(np.trace(M.T@M))

def vec_norm(v):
    return np.sqrt(v.T@v)

def Theta(w=np.array([1,1,2]).reshape(3,1), dt=.1):
    W = Omega(w)
    u = u_b(w, dt)
    W_norm = matrix_norm(W)
    u_b_norm = vec_norm(u)
    return np.cos(u_b_norm/2)*np.eye(4) +
np.sin(u_b_norm/2)/(u_b_norm/2)*W
```

Given the current orientation, the next orientation is obtained via application of the quaternion update matrix;

$$q_{t+1} = \Phi q_t$$

With theta approximated as;

$$\Theta = \cos\left(\frac{|u|}{2}\right) \cdot I + \sin\left(\frac{|u|}{2}\right) \cdot \frac{2\Phi}{|u|}$$

In which;

$$u = \int \omega dt \quad \text{and} \quad \Omega = \begin{bmatrix} -\omega_\times & \omega^T \\ \omega & 0 \end{bmatrix}$$

with

$$\omega_\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

Clustering details

Kalman filtering

- Use prior knowledge of state to inform measurement dependent state update.
- With user inputs of signal and processing noise, the algorithm takes previous states.
- Noise estimates are updated iteratively.

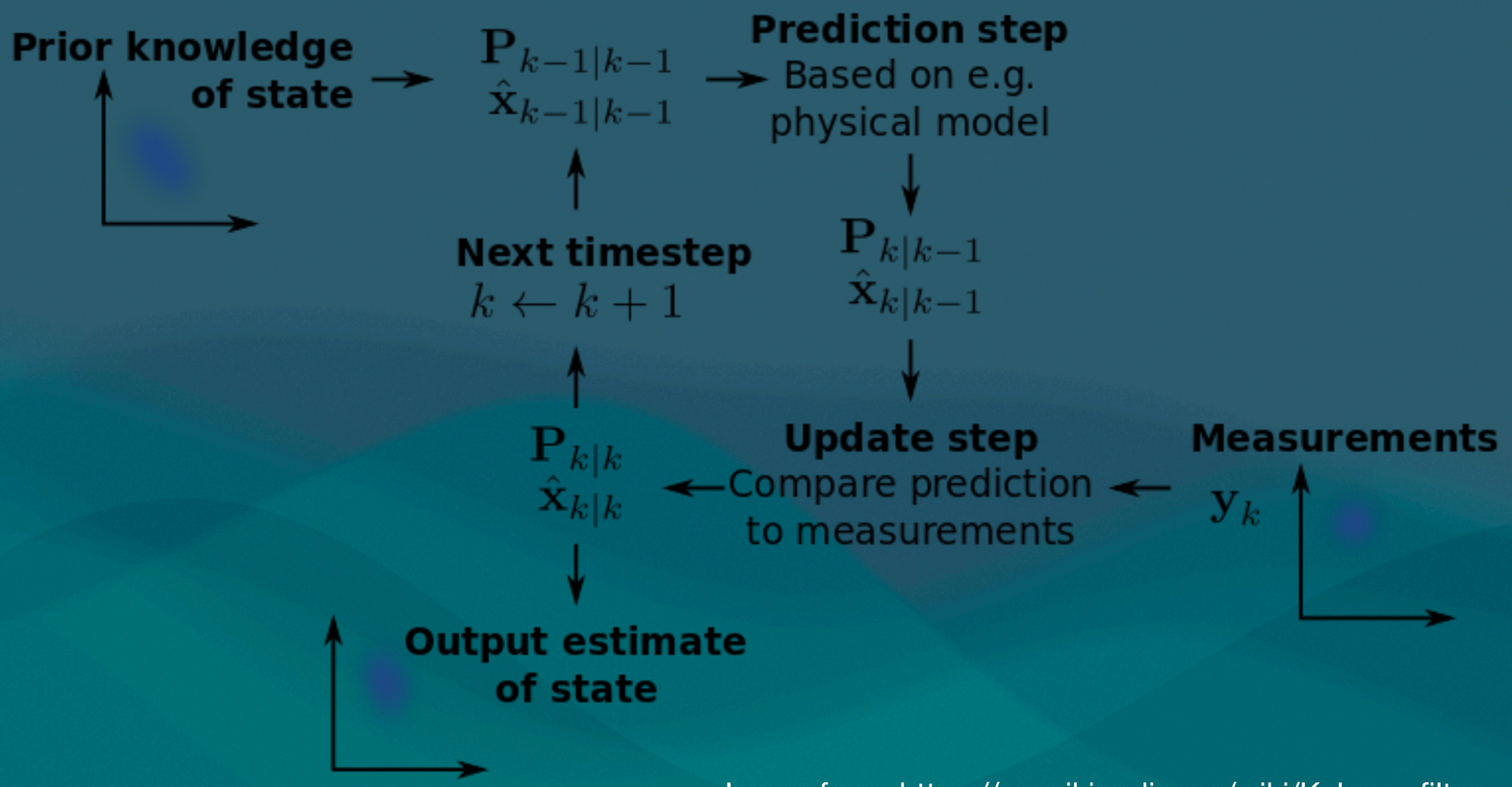


Image from: https://en.wikipedia.org/wiki/Kalman_filter

Does clustering improve performance?

- The 8 different cluster labels were encoded using one hot encoding and used as input features in addition to the IMU data and true orientations for the LSTM
- Each model is trained 3 times on 1.2M data points
- As can be seen below, the inclusion of clustered features did not significantly improve performance, and so we decided not to include them

Av. Performance on test dataset a000_1 (3 trials)	**LSTM**	
Error	With clustering	Without clustering
ATE (m)	12.4 ± 2.7	8.3 ± 2.6
RTE (m)	7.3 ± 3.9	6.6 ± 0.9

List of everything we tried:

Quarternion estimation

- A lot of time was spent on trying to get linear networks, as well as more complicated sequential ones, to predict orientations using just IMU data, as as well a single initial orientation. Most of our models just output constant predictions and were worthless. Our lack of success prompted us to dive into the world of data filters and custom loss functions.
-

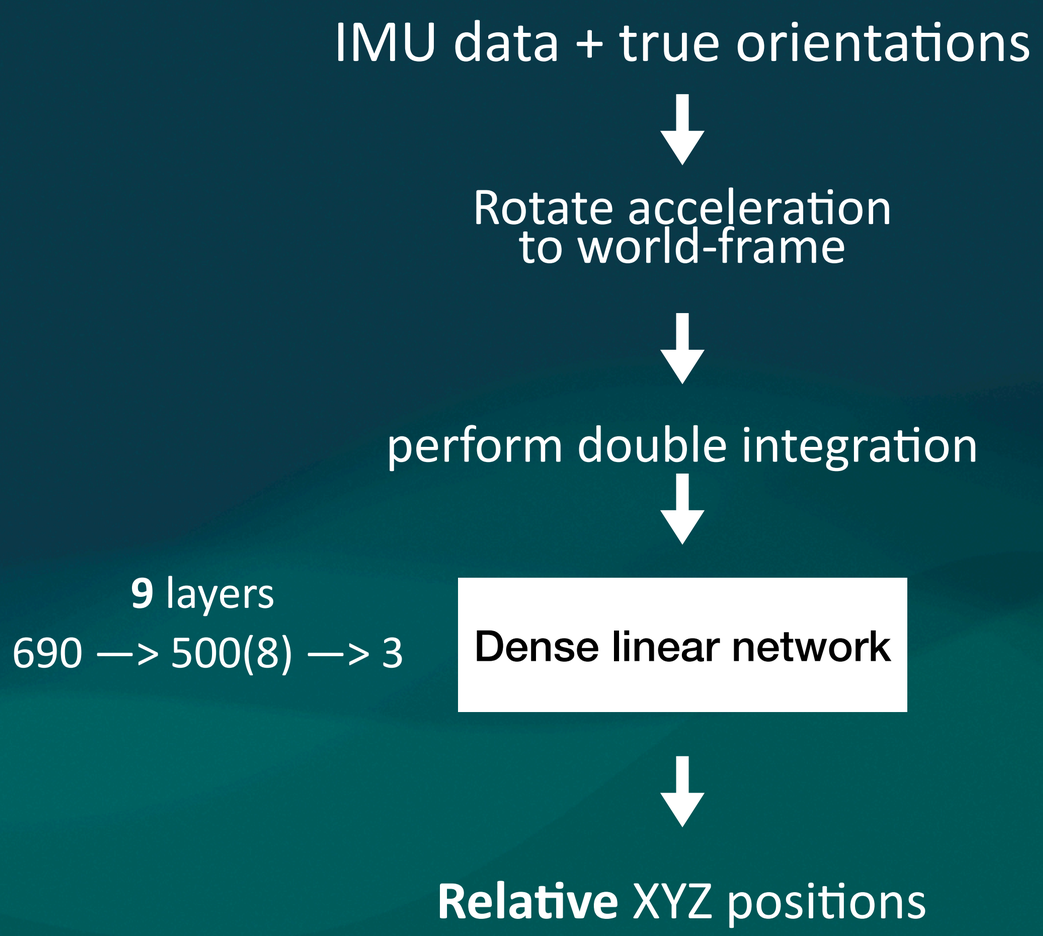
List of everything we tried:

Position estimation using just IMU data

- Much effort was put into bulding a transformer that could predict predictions given solely IMU data, the hope being that this superior architecture would be able to outperform the RoNIN's LSTM model. Transformer based network —> turned out to require more than available compute. The model was unable to learn any relevant structure.

Linear network for position estimation (Pytorch)

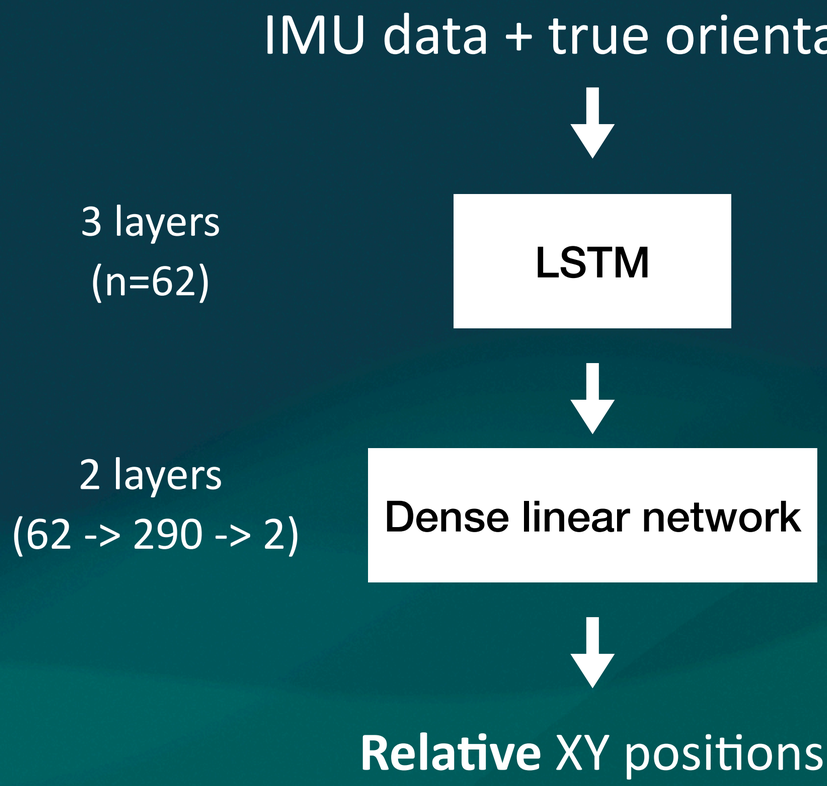
Model summary



Name	Description
Architecture	Densely connected linear
Loss function	MSE
Learning rate	10^{-6}
Sample count (train/val)	1M (80/20)
Runtime	23m 40s
Epochs	500
Sequence length	300
Sequence overlap	1
Batch size	128

LSTM for position estimation (Pytorch)

Model summary

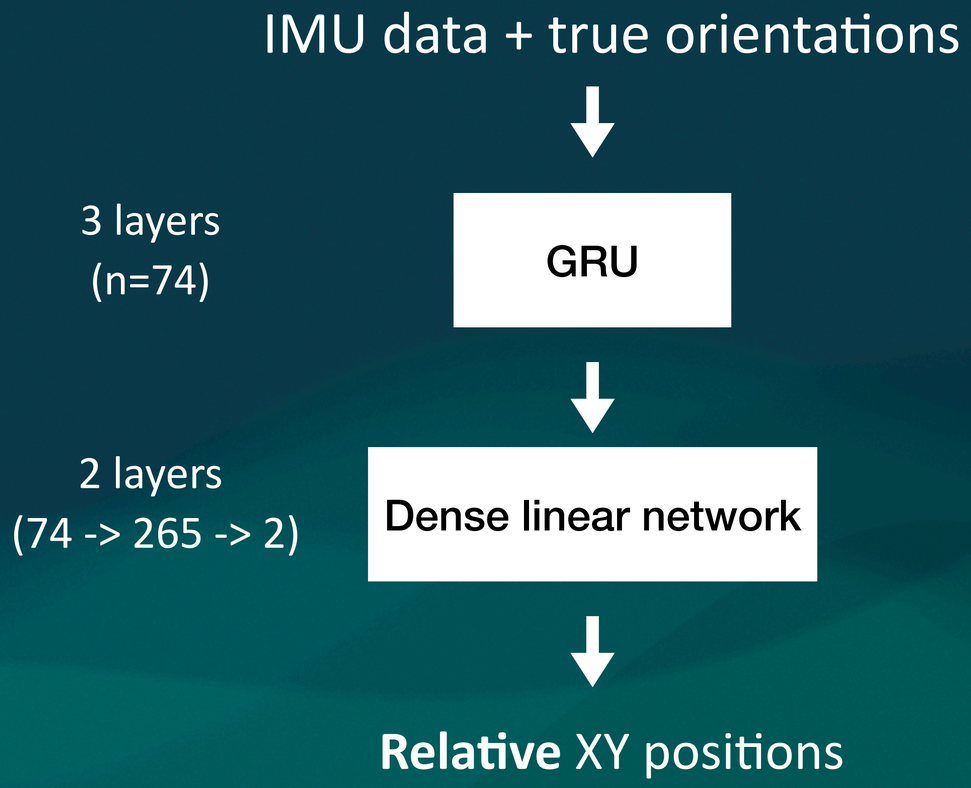


Name	Description
Architecture	LSTM
Loss function	MSE
Learning rate	3.9 * 10^{-5}
Sample count (train/val)	1.2M (80/20)
Runtime	3m 20s
Epochs	78
Sequence length	30
Sequence overlap	1
Batch size	64
Regularization	dropout=0.2

Hyperparameter optimzation: Epochs, sequence length, learning rate and N_hidden were optimized using Optuna for 75 trials (runtime = 5 hours on NVIDIA GeForce 940MX)

GRU for position estimation (Pytorch)

Model summary

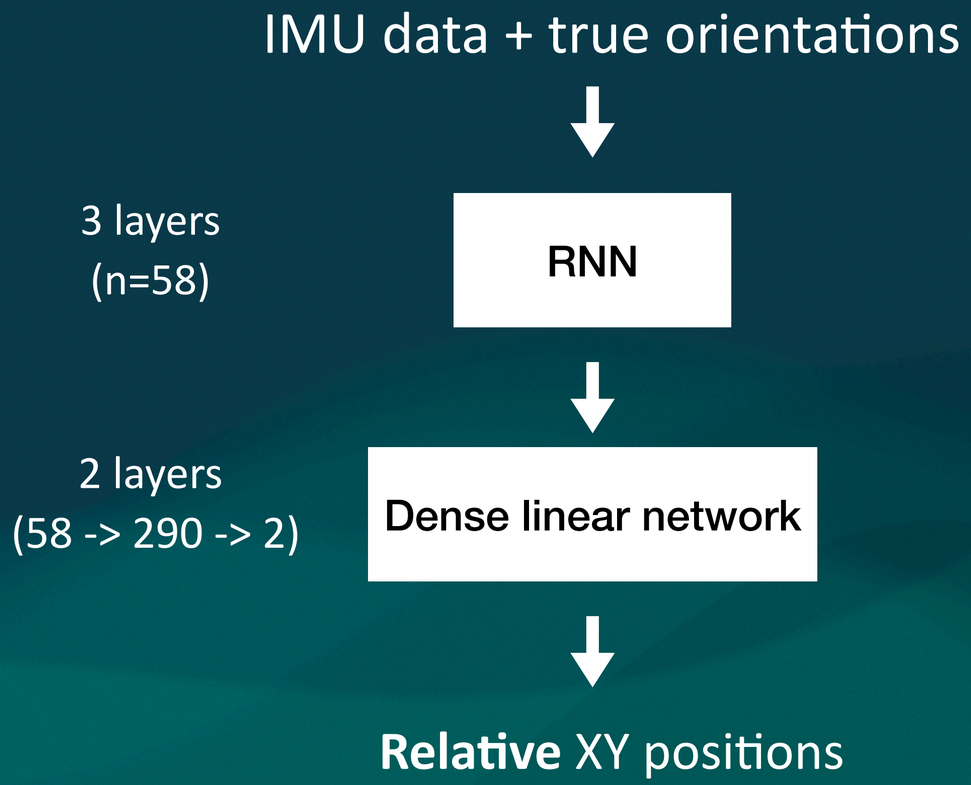


Name	Description
Architecture	GRU
Loss function	MSE
Learning rate	1.7 * 10^{-4}
Sample count (train/val)	1.2M (80/20)
Runtime	5m 10s
Epochs	101
Sequence length	35
Sequence overlap	1
Batch size	64
Regularization	dropout = 0.2

Hyperparameter optimzation: Epochs, sequence length, learning rate and N_hidden were optimized using Optuna for 75 trials (runtime = 8 hours on NVIDIA GeForce 940MX)

RNN for position estimation (Pytorch)

Model summary

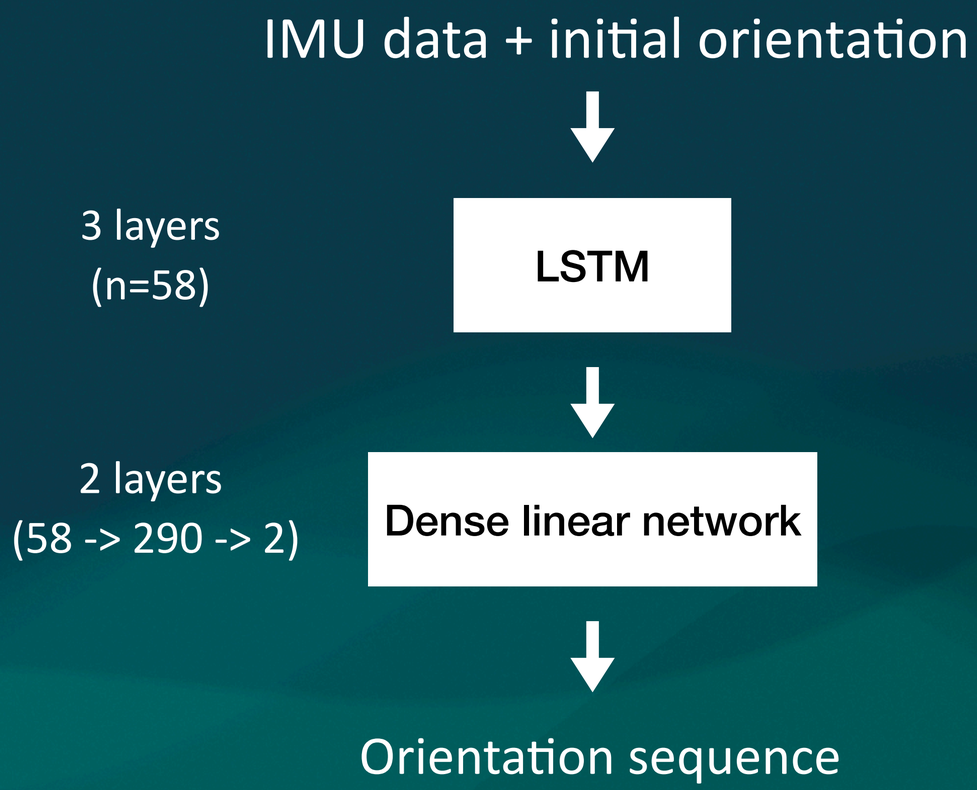


Name	Description
Architecture	RNN
Loss function	MSE
Learning rate	6.44 * 10^{-6}
Number of samples	4.5M
Runtime	
Epochs	101
Sequence length	173
Sequence overlap	20
Batch size	64
Regularization	dropout = 0.25

Hyperparameter optimzation: Epochs, sequence length, learning rate and N_hidden were optimized using Optuna for 20 trials (runtime = 9 hours)

LSTM for orientation estimation (Pytorch)

Model summary



Name	Description
Architecture	LSTM
Loss function	MSE
Learning rate	6.89 * 10^{-6}
Number of samples	4.5M
Runtime	
Epochs	78
Sequence length	322
Sequence overlap	20
Batch size	64

Hyperparameter optimzation: Epochs, sequence length, learning rate and N_hidden were optimized using Optuna for 20 trials (runtime = 9 hours)