Analyzing the energy-consumption of Robust Neural Inertial Navigation

KEVIN FROHBOES, University of Twente, The Netherlands

From fitness trackers to Google maps, positioning is important in a variety of contexts. Many technologies are used for this such as GPS, WiFi or camera-based tracking. GPS and WiFi are however power hungry and all three depend on environmental factors such as lighting and signal availability. Inertial measurements provide a universally applicable alternative to this, their biggest drawback however is a lack of accuracy. A solution for improved accuracy is introduced in the RoNIN-paper. This paper aims to evaluate variations of the RoNIN-architecture for energy efficiency.

Additional Key Words and Phrases: Inertial Navigation, Trajectory, Smartphones, RoNIN

1 INTRODUCTION

Many popular mobile applications such as Google Maps or Pokemon Go rely on position tracking. The same technologies are further used in drones, smaller fitness trackers or autonomous vehicles. Under a clear sky, the accuracy of a smartphone using GPS is typically within 4.9m [14]. This accuracy however degrades with obstacles in the way. WiFi, which has gained notoriety as a tracking solution in recent years, suffers from similar issues of signal availability. Since both solutions depend on radio-communications they also tend to drain battery life. Visual tracking meanwhile raises privacy concerns and is dependent on lighting conditions.

The inertial measurement unit of a smartphone presents in theory an ideal solution to these problems. Smartphone-IMUs are low power and can be used anywhere. The errors in their measurements however accumulate over time degrading positioning quality. These errors can be significantly minimized using deep learning. An influential paper in this area is [9] in which a dataset and an architecture for this problem are introduced. The architecture in the paper (from here on referred to as RoNIN-architecture) comes in three varieties with ResNet18-, LSTM- and TCN-backbones respectively.

While the paper performs an evaluation of the accuracy of the trajectories reconstructed with the architectures they are not deployed on any edge devices (mobile or otherwise). Other aspects of performance, such as resource-usage, are also not considered. The following work aims to correct this by first training the existing and two new varieties of the architecture. And then evaluating their performance on edge devices.

Two experiments are performed to evaluate the architectures. Firstly a power-measurement setup for a Raspberry Pi was created. This allowed measuring inference times and also directly evaluating power consumption. Secondly the architectures were made to process data locally on three smartphones. Here inference time was recorded, which directly relates to power consumption.

TScIT 41, July 5, 2024, Enschede, The Netherlands

2 RELATED WORK

A variety of classical approaches to inertial navigation exist. Pedestrian-dead-reckoning measures steps and uses heading and step-length estimates to deduce position change. For this these systems either use foot mounted sensors (velocity=0 every step) or they infer steps from patterns in IMU data (e.g. via spectral analysis) [7]. Another common approach is double integration of acceleration measurements. This leads to quadratic error growth however. Error can be reduced by introducing motion constraints. One of these constraints is requiring that the IMU regularly come to a stationary position. This means when the IMU measurements are in a range corresponding to being stationary they are set to 0 eliminating the error that accumulates during the stationary phase of walking. A downside of this is that it requires mounting the IMU on a foot.

To deduce and remove higher order error patterns machine learning can be employed. Deep convolutional neural networks for instance enabled significant strides in object-recognition as shown in [16]. Convolutions also allow discovering relations between neighbouring data points in a time series. A larger variety of patterns in data can be deduced by varying the kernel sizes of those convolutions. Stacking kernels is however computationally expensive. Inception addresses this by using several kernels at the same level of a network [18]. Classically deep neural networks suffer from vanishing gradients during training. In [8], residual connections are introduced to address this. To reduce computational costs MobileNets can be applied. In [12] the original MobileNet splits the task of convolution into two lower cost convolutions. This is further modified for performance and complexity in the subsequent iterations of the model [11].

While it is possible to process time-series data using the aforementioned techniques, architectures specifically for time-series do exist. A long-standing approach is using recurrent networks such as LSTM [10]. Convolutional approaches can however outperform recurrence on such tasks as shown in [1]. Temporal Convolutional Networks, which consist of stacked dilated causal convolutions [13], are specifically designed for this task.

An early approach for using machine learning for inertial tracking is [19]. Here an SVM first identifies phone position before two SVRs specific to this positioning regress x- and y-velocity. IONet [2] uses motion data (vector of the last 200 samples of x,y,z-accelerations, roll, pitch and yaw) to deduce the change in position and orientation over the last step. The backbone of this design is LSTM. LSTM was chosen to prevent vanishing/exploding gradients related to RNNs [2]. The architecture is bi-directional as past and future frames are relevant for accurate predictions [2]. To model intermediate steps in determining output vectors, two layers of LSTMs are stacked atop each other.

The RoNIN-architecture is based on this. It has three variations, LSTM, ResNet18 and TCN. Just like IONet, the ResNet-variant takes 200 frames of the same 6D positioning vectors as input. The ResNet architecture has an additional fully-connected layer of 512 units to produce the desired output shape of one 2D velocity-vector. During operation inference is performed every 10 frames. The result is integrated over the time taken to sample the 10 IMU-frames to get position change. The temporal architectures process frames sequentially. For LSTM first a bilinear layer is applied to a frame. Then both the bilinear-output and the input are concatenated. The subsequent LSTM-component consists of three layers with 100 hidden states each. The LSTM output passes two linear layers. Per frame the LSTM-architecture outputs a 2D velocity-vector. The TCN architecture has six blocks with 16, 32, 64, 128, 72, and 36 channels resulting in a receptive field of 253. This is followed by a 1x1 convolution. During operation the 2D-velocity vector for a frame is integrated over the time taken to sample the frame. This yields position change.

For rotation RoNIN uses a heading-agnostic coordinate frame. At inference time all model input and output correspond to a fixed (randomly chosen at the start) coordinate frame where the z-axis aligns with gravity. To achieve this the android "game vector rotation sensor" is used. This utility returns a quaternion which corresponds to rotating positions in the heading-agnostic coordinate frame into the frame aligning with phone position. This heading is used to rotate the linear accelerations which by default are relative to the device. Unlike the "rotation vector sensor", "game vector" does not use geomagnetic measurement to keep its heading, therefore the estimate of device orientation suffers drift over time. An advantage is that this makes measurements immune to magnetic disturbances [6].

Out of all the papers mentioned in [15] only L-IONet ([3]) is concerned with making pedestrian tracking efficient on mobile devices. LSTM which is used in the original IONet suffers from having very complex operations. It is also difficult to paralellize since it operates sequentially. This leads to high resource use and slow inference. In [3] WaveNet is used instead of LSTM, with the softmax-layer replaced by pooling and a fully connected layer to generate position and rotation change-vectors. The L-IONET is deployed on several mobile devices in [3] achieving mean-squared errors comparable to the two-LSTM-structure in IONet, while having significantly faster inference time.

3 PROBLEM FORMULATION

Aside from results on the OxIOD dataset the RoNIN-architecture significantly outperforms IONet. So far there has however not been a paper trying to adapt the design to a limited-resource context in the way [3] did. This paper aims to do so by making two important contributions. An analysis of the performance of RoNIN motion tracking on smartphones and Raspberry Pi and testing a wider range of architecture variations for the backbone of the design. This results in the following Research Questions:

(1) What are the performance characteristics of different variations of the RoNIN architecture?
  (a) How do the architectures introduced in the paper perform (ResNet18, TCN, LSTM)?
  (b) How do MobileNetv3 (Small), VGGNet16 and Inceptionv4 perform?
  (c) What is their inference time on a Smartphone?
  (d) How much energy do they consume on a Raspberry Pi?
  (e) How much error do they generate?

4 EXPERIMENTAL DESIGNS

4.1 Models

Six different backbones were trained: The ResNet18-architecture was taken from [9]. It takes 200x6 input values to produce one 2D-velocity-vector. Inception v3, VGGNet16 and MobileNetv3-small operate with the same in- and outputs. All operations that would regularly be 2D were converted to their 1D counterparts when building the models.

To reduce training time the number of inception blocks was reduced from (InceptionA: 4, InceptionB: 7, InceptionC: 3) to (InceptionA: 1, InceptionB: 2, InceptionC: 1). For Inception v4, the final softmax and dropout were replaced by FCOutputModule taken from model_resnet1d.py. This module consists of three linear, two ReLU and two dropout layers. FCOutputModule allows changing the output to the appropriate shape. Dropout was 0.2. The Trans_planes argument was set to None as having more planes in the later layer would add more parameters. For VGGNet the final fully connected layers were likewise replaced by the FCOutputModule. Here dropout was 0.5 like for ResNet18. LSTM and TCN were used in the same way as in the RoNIN-paper [9].

For mobile deployment the models were made more compact using "torch.utils.optimize_for_mobile". This utility fuses different types of operations together to reduce model size [5].

Table 1. Number of parameters in the different architectures

Model	Num. Parameters
ResNet18	4634882
VGGNet	5627266
MobileNet	1473106
Inception	10143938
TCN	540488
LSTM	205832

4.2 Methodology

4.2.1 Training. The authors of [9] made pre-trained versions of their architectures available on GitHub. These were not used and instead all six backbones were trained with the same data. This was necessary as only part of the RoNIN-dataset was published meaning otherwise a proper comparison among architectures would have been impossible. The same training methods as in [9] were used. The ResNet18, VGGNet and MobileNet architectures were trained to predict positional change over the 200 input frames. This training employed mean-squared-error loss. In the temporal case the outputs over 253 (LSTM) and 400 (TCN) frames were summed. During training the L2-norm of this sum and position change over the frames was minimized. In [9] this unexpectedly lead to predicting instantaneous velocity per frame. Training was performed using an NVIDIA Tesla T4 with 256 GB of memory.

Training was considered complete upon stagnation of the validation error. In the case of VGGNet and ResNet this occurred after 39 iterations. For MobileNet 18 iterations of training were deemed sufficient. TCN took 42 iterations and LSTM 16. Inception did not converge during training.

4.2.2 Dataset. Only about half of the data used in [9] was made publicly available. Of this only the RoNIN-dataset was used here, as it was sufficiently large. The data was further split into three sets: Training, Validation and Testing. Due to the significantly slower training process of LSTM and TCN, a subset of the training set was used for them.

4.2.3 Evaluation. To test accuracy the models were run on the test set with code to reconstruct trajectories from their predictions. The reconstructed trajectories can be seen under A. In [9] two metrics Absolute Trajectory Error (ATE) and Relative Trajectory Error (RTE) are used. As per [17], ATE is calculated by first rotating and translating the trajectory to minimize root-mean-square error with ground-truth. ATE is the root-mean-square error that then results from this. RTE is the root-mean-square error over a fixed (here one minute) time-interval. The same metrics were used during evaluation.

The Python code used for reconstructing trajectories was run on a Raspberry Pi for all six models and all four test files. Using the Arduino power measurement setup in 1 power draw was logged. The python code records the time it takes to run, excluding the time to load the model and dataset. Using operating time and power measurements, total energy use was calculated. The models were

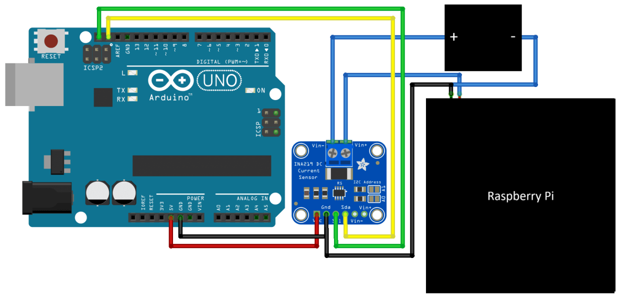


Fig. 1. Wiring diagram. The diagram is a modified version of a diagram on the website in [4]. An INA219 was used to measure power draw. The device is embedded in a JZK CJMCU-219 PCB (blue chip in the middle of the schematic).

further deployed on three mobile devices: a Samsung Galaxy S8, a Samsung Galaxy S23 and an HTC U11. A program was written in Android Studio to perform inference on the same testing data as used on the Raspberry Pi. Trajectory reconstruction was not implemented locally to avoid complexity.

5 RESULTS AND DISCUSSION

5.1 Accuracy

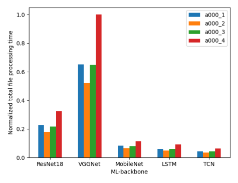
The error of the different models can be seen in 2. Inception did not converge during training and is therefore showing high levels of trajectory error. This is likely due to vanishing gradients as the model contains 51 subsequent convolutions. Due to not converging it was excluded from the experiments. Besides for LSTM, error performance of the remaining architectures was better than for the tests on unseen data performed in [9]. This can be explained by the fact that only a subset of the testing data from [9] was used here. Non-temporal architectures outperformed temporal ones. A likely cause is less training data having been used compared to VGGNet, ResNet18 and MobileNet. Lower parameters counts (see 1) might play a role too. As per [1] it was expected that LSTM underperforms against TCN. This likely also relates to it having less than half as many parameters (see 1). VGGNet performs best in terms of ATE. Here too higher network complexity (see 1) offers an explanation.

Table 2. The average ATE and RTE (in m) over the 4 trajectories in the test-dataset

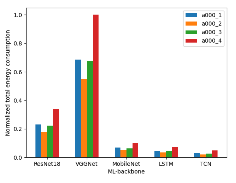
Model	**ATE**	**RTE**
ResNet18	3.1	2.1
VGGNet	2.1	2.2
MobileNet	3.1	2.0
Inception	15.4	20.2
LSTM	5.0	2.6
TCN	3.1	2.3

5.2 Power consumption

As can be seen below the total energy consumption and inference time closely correlated indicating that inference time was a good proxy for energy use. Relative energy draw of LSTM and TCN was very slightly lower compared to inference time. Due to their quick inference times and uncertainty related to the start of measurements this can be attributed to measurement error.



(a) Normalized total time for processing the four test files on Raspberry Pi.



(b) Normalized energy draw for processing the four test files on Raspberry Pi.

Fig. 2. Comparison between normalized energy use and inference time on Raspberry Pi

5.3 Inference Time

On the Raspberry Pi ResNet18 was faster than VGGNet. This is in line with lower parameter counts. ResNet was in turn outperformed by MobileNet, which operates with less computationally intensive convolution operations. The inherently temporal LSTM and TCN significantly outperformed MobileNet, with TCN performing better than LSTM. This is in spite of having about twice as many parameters 1.

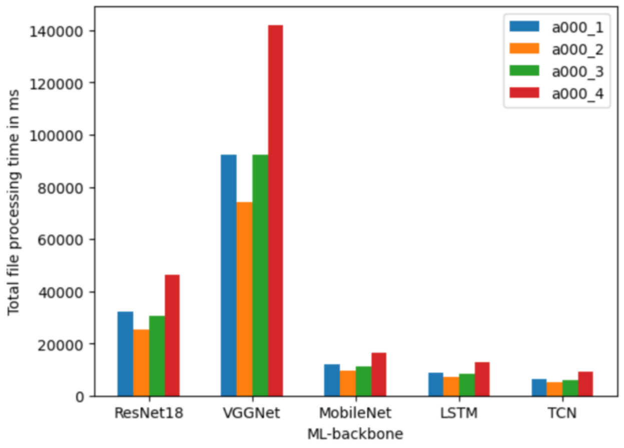


Fig. 3. Total times (in ms) for processing the four test files on Raspberry Pi.

Surprisingly, unlike on Raspberry Pi, Resnet18 consistently underperformed compared to VGGNet on all three smartphones. On the S20 and U11 MobileNet was still the fastest non-temporal architecture. LSTM and TCN still significantly outperformed the other architectures on the phones.

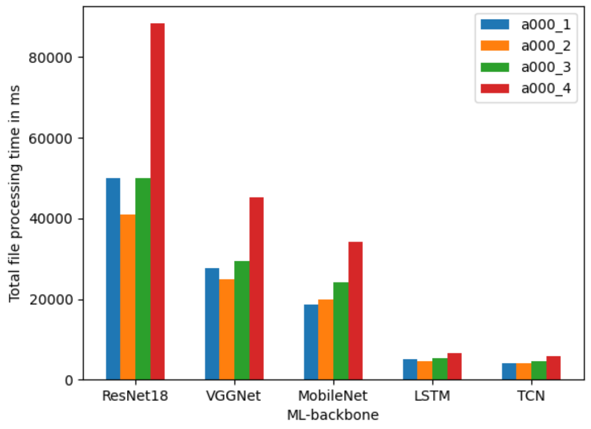


Fig. 4. Total inference times (in ms) for processing the 4 test files on a Samsung Galaxy S23

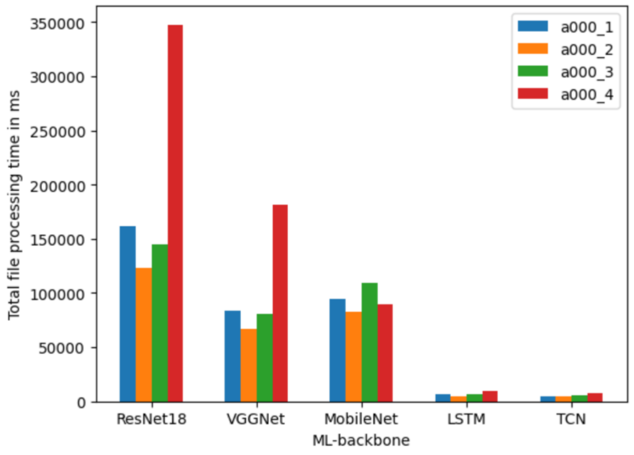


Fig. 5. Total inference times (in ms) for processing the 4 test files on a Samsung Galaxy S8

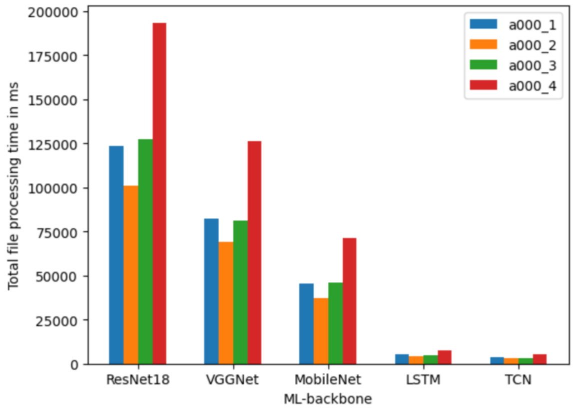


Fig. 6. Total inference times (in ms) for processing the 4 test files on an HTC U11

The code was also run without any ML-inference, purely loading and storing data. The non-temporal data processing approach took significantly more time here. For non-temporal out of a set of N-frames, samples of 200 frames are taken at every 10th frame. This results in roughly 200 * N/10 = 20*N data points for processing. This stands in contrast to merely processing N-data points in the temporal case. The difference in pure loading- and storing-time between temporal and non-temporal was significant. It was not 20-fold, likely due to fixed costs for loading data. The measured inference times in 3, 4, 5, 6 had differences in excess of factor 20. From this it can be concluded that even if the temporal and non-temporal architectures were to process the same amount of data the temporal design would still be faster.

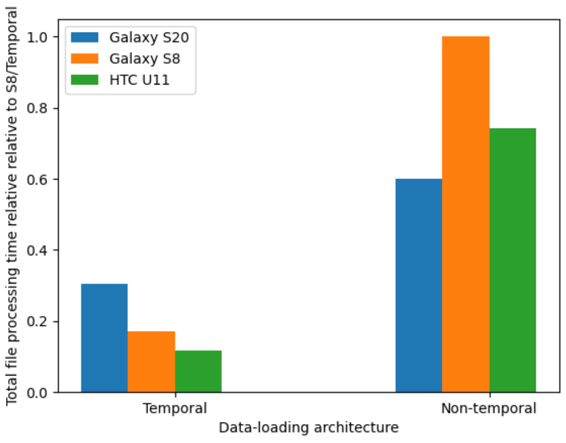


Fig. 7. Average file data loading times over the 4 files. The times are normalized against the highest loading time (highest loading time corresponds to non-temporal + HTC U11 in each case).

5.4 Flaws

Processing a test file on the phones does not directly mirror real-world use of the devices. Under real-world conditions IMU-data would regularly be sampled and used to continuously calculate position. Since this results in only short bursts of model usage it was not deemed suitable for measuring model inference times. The phone test does not include reconstruction of the path it merely generates the data needed to do so. This might skew results since for N-data points the non-temporal architectures generate about N/10 outputs as opposed to N-many in the temporal case. Given that the Raspberry Pi has path-reconstruction and the trend of temporal-strongly outperforming non-temporal-architectures persists, this is unlikely to result in large alterations of experiment outcomes.

In some cases the time spent on temporal-inference is negligible compared to data loading. For example operation time for TCN on the Galaxy S23 was 6036ms. Running a version of the program that performs no inference took 5973ms (the difference amounts to random fluctuations). This makes it difficult to compare temporal architectures.

To allow for faster training only a subset of the training data was used for LSTM and TCN. This casts doubt over the correctness of accuracy-comparisons made between temporal and non-temporal architectures.

6 CONCLUSIONS

It can be seen that TCN is the ideal solution for minimizing energy consumption. Further inherently temporal architectures represent a more energy efficient approach to IMU processing. 2 reasons for this were found. For one the temporal mode of operation processes data points only once instead of processing a sliding window (which contains repeats). Secondly the temporal architectures themselves process the same amount of datapoints faster. When targeting maximal accuracy the high performance of VGGNet indicates that greater parameter counts still allow for accuracy gains over the architectures presented here. Directly measuring power draw was shown unnecessary as inference time closely correlates with it.

ACKNOWLEDGMENTS

I would like to thank Nguyen, Minh Son for supervising this work, providing valuable ideas, insights and materials to it.

REFERENCES

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. 2018. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. arXiv:1803.01271
[2] Changhao Chen, Xiaoxuan Lu, Andrew Markham, and Niki Trigoni. 2018. IONet: Learning to Cure the Curse of Drift in Inertial Odometry. *CoRR* abs/1802.02209 (2018). arXiv:1802.02209 http://arxiv.org/abs/1802.02209
[3] Changhao Chen, Peijun Zhao, Chris Xiaoxuan Lu, Wei Wang, Andrew Markham, and Niki Trigoni. 2020. Deep-Learning-Based Pedestrian Inertial Navigation: Methods, Data Set, and On-Device Inference. *IEEE Internet of Things Journal* 7, 5 (2020), 4431–4441. https://doi.org/10.1109/JIOT.2020.2966773
[4] elosciloscopio.com. 2024. *Tutorial INA219 para Arduino, ESP8266 y ESP32.* https://elosciloscopio.com/tutorial-ina219-arduino-esp8266-y-esp32/
[5] The Linux Foundation. 2024. *torch.utils.mobile_optimizer.* https://pytorch.org/docs/stable/mobile_optimizer.html
[6] Google. 2024. *Motion Sensors.* https://developer.android.com/develop/sensors-and-location/sensors/sensors_motion#sensors-motion-rotate
[7] Robert Harle. 2013. A Survey of Indoor Inertial Positioning Systems for Pedestrians. *IEEE Communications Surveys Tutorials* 15, 3 (2013), 1281–1293. https://doi.org/10.1109/SURV.2012.121912.00075
[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385
[9] Sachini Herath, Hang Yan, and Yasutaka Furukawa. 2020. RoNIN: Robust Neural Inertial Navigation in the Wild: Benchmark, Evaluations, New Methods. In *2020 IEEE International Conference on Robotics and Automation (ICRA).* 3146–3152. https://doi.org/10.1109/ICRA40945.2020.9196860
[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[11] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. arXiv:1905.02244
[12] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861
[13] Colin Lea, Michael D. Flynn, Rene Vidal, Austin Reiter, and Gregory D. Hager. 2016. Temporal Convolutional Networks for Action Segmentation and Detection. arXiv:1611.05267
[14] Navigation National Coordination Office for Space-Based Positioning and Timing. 2022. *GPS Accuracy.* https://www.gps.gov/systems/gps/performance/accuracy/
[15] Swapnil Sayan Saha, Sandeep Singh Sandha, Luis Antonio Garcia, and Mani Srivastava. 2022. TinyOdom: Hardware-Aware Efficient Neural Inertial Navigation. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 2, Article 71 (jul 2022), 32 pages. https://doi.org/10.1145/3534594

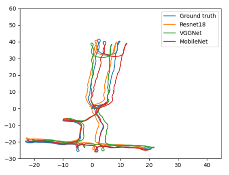
[16] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556

[17] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. 2012. A benchmark for the evaluation of RGB-D SLAM systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 573–580. https://doi.org/10.1109/IROS.2012.6385773

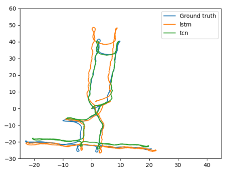
[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. arXiv:1409.4842

[19] Hang Yan, Qi Shan, and Yasutaka Furukawa. 2017. RIDI: Robust IMU Double Integration. *CoRR* abs/1712.09004 (2017). arXiv:1712.09004 http://arxiv.org/abs/1712.09004

A APPENDIX TRAJECTORIES

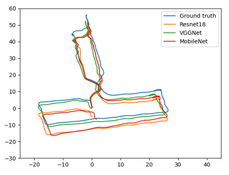


(a) Trajectories for non-temporal architectures

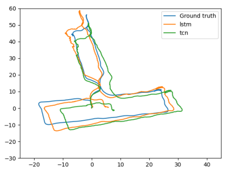


(b) Trajectories for temporal architectures

Fig. 8. Trajectories reconstructed with the models for test file a000_1

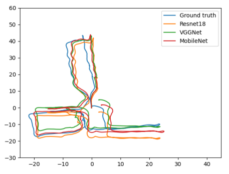


(a) Trajectories for non-temporal architectures

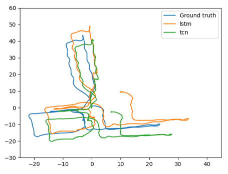


(b) Trajectories for temporal architectures

Fig. 9. Trajectories reconstructed with the models for test file a000_2

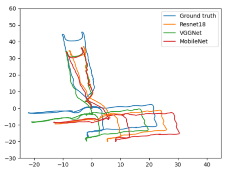


(a) Trajectories for non-temporal architectures

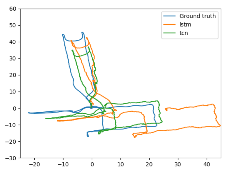


(b) Trajectories for temporal architectures

Fig. 10. Trajectories reconstructed with the models for test file a000_3



(a) Trajectories for non-temporal architectures



(b) Trajectories for temporal architectures

Fig. 11. Trajectories reconstructed with the models for test file a000_4