RBE 549:
P4: Deep and Un-Deep VIO

Deepak Harshal Nagle
Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609
Email: dnagle@wpi.edu
Telephone: (774) 519-8335

Irakli Grigolia
Computer Science
Worcester Polytechnic Institute
Worcester, Massachusetts 01609
Email: igrigolia@wpi.edu
Telephone: (508) 373-3402

***Abstract*—This report presents two approaches for implementing stereo visual-inertial odometry algorithms: the classical Multi-State Constraint Kalman Filter (MSCKF) and a deep learning-based method for enhanced performance. The implementation builds on a pre-existing codebase provided as starter code. The paper details the mathematical concepts underlying the stereo-MSCKF algorithm for the classical approach and the development of a novel deep learning architecture for visual-inertial fusion as an alternative.**

The results and observations for each approach are documented in this report. For the classical approach, we focus on the MSCKF algorithm's implementation and performance. The deep learning approach explores the integration of a deep learning component into the visual-inertial odometry system. The effectiveness of both methods is analyzed through experiments and comparisons with ground truth trajectories, providing insights into the potential of each approach for improving visual-inertial navigation systems.

INTRODUCTION

In this project, we investigate two separate approaches for estimating depth from images by obtaining scale. The first approach employs a classical method, which combines stereo cameras and Inertial Measurement Units (IMUs) for pose estimation and depth backtracking. Stereo cameras can estimate depth by matching features but face computational expense and motion blur limitations. IMUs are effective in fast movements and jerks but suffer from drifts over time. To tackle these challenges, we utilize a filter-based stereo visual-inertial odometry algorithm using the MultiState Constraint Kalman Filter (MSCKF) and test it on the Machine Hall 01 easy subset of the EuRoC dataset.

In the second approach, we examine the potential of deep learning techniques for enhancing the performance of visual-inertial odometry systems. We develop a novel deep learning architecture for visual-inertial fusion, independent of the classical approach. For each method, we analyze the results by comparing them to the ground truth in order to assess their potential for improved depth estimation and navigation systems. This evaluation allows us to better understand the strengths and limitations of each approach and their respective contributions to the field of visual-inertial navigation.

I. PHASE 1: UN-DEEP VISUAL INERTIAL ODOMETRY

DATA

Machine Hall 01 easy subset of the EuRoC dataset is used to test the implementation. The data is collected using a VI sensor carried by a quadrotor flying a trajectory. The ground truth is provided by a sub-mm accurate Vicon Motion capture system.

FUNCTIONS

We implement Multi-State Constraint Kalman Filter (MSCKF), some of the functions implemented are described below.

initialize_gravity_and_bias:

This function initializes the initial orientation and bias based on the first readings from the IMU. The average angular and linear velocities are calculated from the IMU message buffer's first few readings. The gyro bias is initialized using the average angular velocity, and gravity is calculated using the linear acceleration. The normalized gravity vector is used as the IMU state, and the initial orientation is set consistently with the inertial frame. The quaternions represent the final vector, where ${}^{\mathrm{G}}_{\mathrm{I}}\mathbf{q}$ denotes the rotation from the inertial frame to the body frame, which in this case is the IMU frame. The vectors ${}^{\mathrm{G}}\mathbf{v}_{\mathrm{I}}$ and ${}^{\mathrm{G}}\mathbf{p}_{\mathrm{I}}$ represent the body frame's velocity and position in the inertial frame, and $\mathbf{b}_{\mathrm{G}}$ and $\mathbf{b}_{\mathrm{a}}$ are the biases of the measured angular and linear velocities from the IMU. The representation of the final vector is given by the following expression

$$X_I = ({}^I_G\mathbf{q}^T \;\; \mathbf{b}_g^T \;\; {}^G\mathbf{v}_I^T \;\; \mathbf{b}_a^T \;\; {}^G\mathbf{p}_I^T \;\; {}^I_C\mathbf{q}^T \;\; {}^I\mathbf{p}_c^T)^T$$

batch_imu_processing:

The function deals with processing IMU messages from a buffer, taking into account a specified time range. It operates by running the process model for each IMU input that falls within the time range, repeating this process until the end of the range is reached. Once completed, the current IMU ID is

updated to the next state ID, and any unused IMU messages are removed from the buffer.

process_model:

The aim of this function is to determine the camera module's pose (dynamics) based on the most recent update of the IMU state. To achieve this, the function takes in the time, m_gyro (current angular velocity), and m_acc (current linear acceleration) as arguments. Following this, the function calculates the error for each IMU state, as represented by the following formula.

$$\tilde{X}_I = ({}^I_G\tilde{\theta}^T \;\; \tilde{\mathbf{b}}_g^T \;\; {}^G\tilde{\mathbf{v}}_I^T \;\; \tilde{\mathbf{b}}_a^T \;\; {}^G\tilde{\mathbf{p}}_I^T \;\; {}^I_C\tilde{\theta}^T \;\; {}^I\tilde{\mathbf{p}}_c^T)^T$$

The linearized continuous dynamics for the error IMU state is calculated as follows:

$$\dot{\tilde{X}}_I = F\tilde{X}_I + Gn_I$$

F and Q (discrete transition matrices) are calculated as follows:

$$F = \begin{bmatrix} -\lfloor\hat{\omega}_\times\rfloor & -I_3 & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ -C({}^I_G\mathbf{q}^T)\lfloor\hat{a}_\times\rfloor & 0_{3\times3} & 0_{3\times3} & -C({}^I_G\mathbf{q}^T) & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & I_3 & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \end{bmatrix}$$

$$G = \begin{bmatrix} -I_3 & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & I_3 & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & -C({}^I_G\mathbf{q}^T) & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & I_3 \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \end{bmatrix}$$

The matrix exponential is approximated to third order as follows:

$$\phi = I_{21\times21} + F(\tau).d\tau + \frac{1}{2} * (F(\tau).d\tau)^2 + \frac{1.0}{6.0} * (F(\tau).d\tau)^3$$

predict_new_state:

After obtaining the current state, we apply the fourth-order Runge-Kutta method to propagate the state and predict its new value. Specifically, we use a function called "predict new state," which takes as input the time step (dτ), the gyroscope data, and the acceleration information for the current state. To begin, we calculate the normalized error state of the angular velocity data. Next, we compute the Ω matrix by following a specific procedure.

$$\Omega(\hat{\omega}) = \begin{bmatrix} -\lfloor\hat{\omega_\times}\rfloor & \omega \\ -\omega^T & 0 \end{bmatrix}$$

$$k1 = f(t_n, y_n)$$

$$k2 = f(t_n + \frac{d\tau}{2}, y_n + k1 * \frac{d\tau}{2})$$

$$k3 = f(t_n + \frac{d\tau}{2}, y_n + k2 * \frac{d\tau}{2})$$

$$k4 = f(t_n + d\tau, y_n + k3 * d\tau)$$

We retrieve the present orientation, velocity, and position data from the IMU state server. Based on the current state and the Ω values, we compute the angular velocity and acceleration, which we then approximate using the Runge-Kutta method.

Once we have computed the estimated orientation, we convert it into quaternions and use this information to update the velocity and position data for the current IMU state. These updated values are then used as the current state information to determine the next state in the sequence.

State_Augmentation:

In this step, our aim is to calculate the state covariance matrix, which will help us in disseminating the ambiguity of the given state. Initially, we extract the IMU and camera state values that correspond to the rotation from the IMU to the camera and the translation vector from the camera to the IMU. Subsequently, we incorporate a fresh camera state into the state server, utilizing the initial IMU and camera state. The augmentation Jacobian is calculated as follows:

$$J_I - \begin{bmatrix} C({}^I_G\hat{\mathbf{q}}) & 0_{3\times9} & 0_{3\times3} & I_3 & 0_{3\times3} \\ -C({}^I_G\hat{\mathbf{q}})^T\lfloor{}^I\hat{\mathbf{p}}_{c\times}\rfloor & 0_{3\times9} & I_3 & 0_{3\times3} & I_3 \end{bmatrix}$$

The state covariance matrix is resized and the IMU state is propagated:

$$P_{k+1|k} = \begin{bmatrix} P_{II_{k+1|k}} & \phi_k P_{IC_{k|k}} \\ P^T_{IC_{k|k}}\phi^T_k & P_{CC_{k|k}} \end{bmatrix}$$

$$P_{k|k} = \begin{bmatrix} J_{21+6N} \\ J \end{bmatrix} P_{k|k} \begin{bmatrix} J_{21+6N} \\ J \end{bmatrix}^T$$

This is the Augmented covariance matrix. It is regularly updated through this function.

F. Incorporating feature observations

Firstly, the feature message is acquired as input for this function. We then determine the current IMU state ID and evaluate the number of features. Following that, we successively add each feature from the feature message to the map server if it isn't already present. Additionally, we maintain a count of the tracked features. For every given state, the map server is updated, and all features are tracked. The tracking rate is computed as the ratio of the number of tracked features to the total number of available features.

G. Updating measurements

A measurement model is utilized to update state estimates. A residual, denoted as r, is linearly dependent on state errors, represented by the following relation [3]:

$$\mathbf{r} = \mathbf{H}\tilde{\mathbf{X}} + noise$$

In the equation above, H represents the measurement Jacobian matrix, and the noise term denotes a zero-mean, white, uncorrelated state error. The estimated Kalman filter framework is implemented. Initially, we examine whether the existing H and r values are zero. We then attempt to reduce the complexity of the Jacobian matrix using QR decomposition to minimize computation requirements:

$$H_x = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} T_h \\ 0 \end{bmatrix}$$

Here, Q1 and Q2 are unitary matrices with columns that form bases for the range and null space of H_x., respectively. T_H is an upper triangular matrix. Next, we calculate the Kalman gain according to the equation:

K = P T_H $(T_H P {T_H}^T + R_n)^{-1}$

In this equation, K represents the Kalman gain, P is the state covariance matrix,T_H^T is the upper triangular matrix, and R_n is the noise covariance matrix. After calculating the Kalman gain, the state error is computed as:

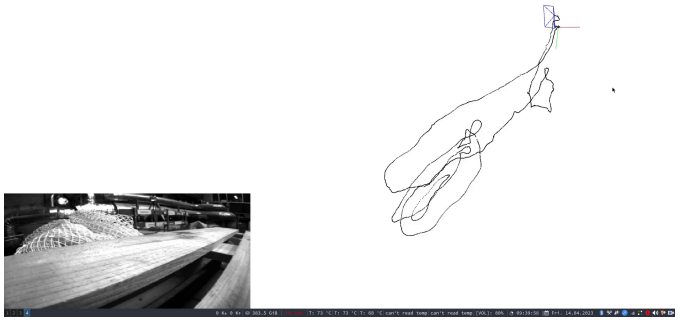
ΔX = K r_n

Using this state error, the IMU state is updated first, followed by the camera states. Lastly, the state covariance is updated, and the covariance matrix is adjusted to be symmetric.

IV. Results

The input data employed for this project is sourced from the Machine Hall 01 easy (MH 01 easy) subset of the EuRoC dataset. Figure 1 displays the trajectory output for this data, which aligns with the anticipated outcome. The output video is included with the code files.

Final plot:



II. Phase 2: Deep Visual Inertial Odometry

III. Introduction

In Phase 2 of this project, we investigate the potential of deep learning techniques for improving visual-inertial odometry systems. We develop a deep learning architecture for visual-inertial fusion by implementing three different networks: Visual Odometry (VO), Inertial Odometry (IO), and Visual-Inertial Odometry (VIO). The VO network uses only visual information, the IO network uses only inertial measurements, and the VIO network combines both visual and inertial data. These networks are trained to predict the relative pose between two image frames along with IMU measurements.

We adapt and modify existing deep learning architectures such as DeepVO and VIONet.The IMU data is preprocessed and fused with visual information to improve the accuracy of the predictions. We use a custom loss function to optimize the network parameters for the best performance.

The effectiveness of the deep learning approach is evaluated by comparing the results with the ground truth trajectory. This analysis provides insights into the strengths and limitations of the VO, IO, and VIO networks for visual-inertial fusion. The potential benefits of incorporating deep learning techniques for improving the accuracy of visual-inertial odometry systems are discussed.

A. Data:

We trained our neural network using the Machine Hall 02 easy (MH_02_easy), Machine Hall 03 medium (MH_03_medium), Machine Hall 04 difficult ($MH_04_difficult$), and Machine Hall 05 difficult ($MH_05_difficult$) subsets of the EuRoC dataset. We tested our model on the Machine Hall 01 easy (MH_01_easy) subset of the dataset.

B. Related Work:

We started our project by performing a thorough analysis of existing networks that perform well for specific datasets.

One of the pioneering works in this domain is DeepVO, proposed by Wang et al. [1], which utilized Convolutional

Neural Networks (CNN) to extract features from image sequences and Recurrent Neural Networks (RNN) to predict the pose incrementally. DeepVO demonstrated a significant improvement over classical approaches, but it relied heavily on supervised training and large amounts of labeled data.

Clark et al. [2] introduced VINet, which combined the strengths of CNNs for image feature extraction and LSTMs for modeling temporal dependencies in IMU measurements. VINet showed promising results in terms of robustness and accuracy but was computationally expensive due to the separate networks for visual and inertial data processing.

In summary, various deep learning architectures have been proposed for VIO, such as CNNs for visual feature extraction and RNNs or LSTMs for modeling temporal dependencies in inertial data. While these methods show promising results in terms of accuracy and robustness, they often suffer from drawbacks such as the need for large amounts of labeled data, computational complexity, scale ambiguity, or limited applicability.

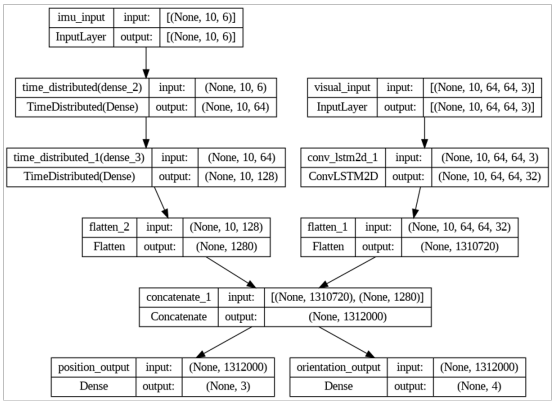


Fig. 1. Our TCN Network

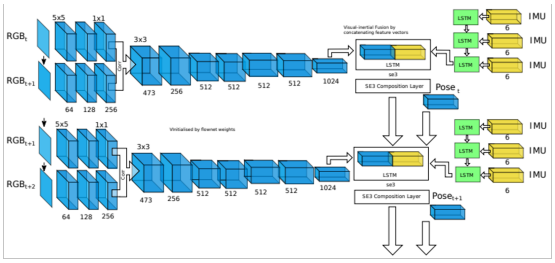


Fig. 2. VINet

C. Data Pre-processing:

The timestamps of images, IMU data, and ground truth data were not initially aligned. We first addressed the misalignment between the IMU data and ground truth data, which had more than 75% overlap. We read the timestamps from their respective CSV files and removed the timestamps that did not match between the two datasets. This resulted in two CSV files (ground truth and IMU data) with matching timestamps.

Next, we tackled the issue of image data timestamps, which had a rate of roughly 1/10th compared to IMU or ground truth data and did not align with them. We resolved this by interpolating the IMU and ground truth data to obtain their values at the timestamps corresponding to the image data. During this process, we removed outlier image timestamps that were not bound between two IMU/ground truth values and deleted the corresponding images. We then performed interpolation for the remaining image timestamps using the closest IMU/ground truth timestamps and their weighted averages, with weights based on their proximity to the image timestamps.

After completing the interpolation, we discarded all IMU and ground truth data except for the timestamps corresponding to each image timestamp. We performed these preprocessing steps for all five datasets. For training, we combined MH02, MH03, MH04, and MH05 into a single file in the same sequence, allowing us to train the model using all of them. We utilized the timestamp values in the three CSV files to track, read, and loop through the input data during training.

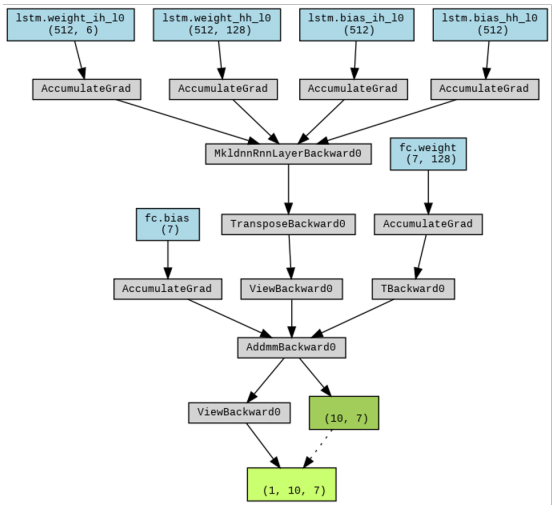


Fig. 3. Our IO Network

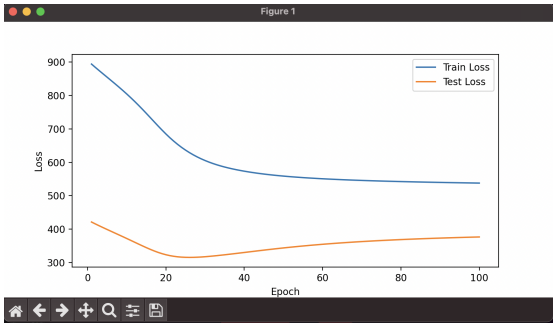


Fig. 4. IO Loss Plot

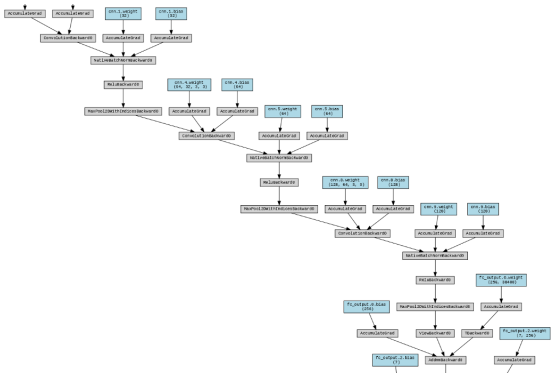


Fig. 5. Our VO Network

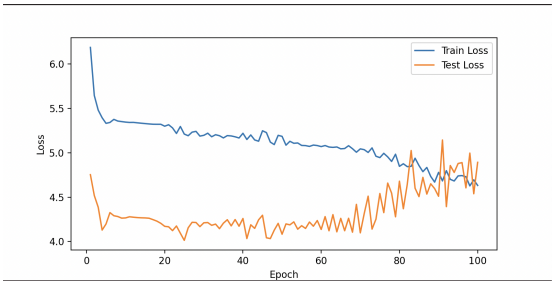


Fig. 6. VO Loss Plot

D. Our Approach:

In our research of deep learning architectures for visual-inertial odometry systems, we found that most existing networks utilizing sequential data relied on LSTMs and similar networks, which proved to be data-hungry and slow to train. After consulting with our professor, we began exploring networks based on temporal CNNs as a more efficient starting point.

We developed our own architecture for VIO, which involved time-distributed layers and was designed to be split into separate VO and IO networks after training. The network was trained using normalized stacked stereo-images as a visual input and IMU data as an inertial input.

Our approach was to create a complete VIO network, train and deploy it, and then split it into separate VO and IO parts. The VIO architecture we developed is illustrated in the diagram below:

After initial experiments with temporal CNNs, we found that our implementation was slow due to the large amount of images involved (around 10,000 from five datasets). Running the network on our CPU was taking a significant amount of time, which led us to consider alternative approaches for improving its speed and efficiency.

One idea we came up with was to use regular CNNs for processing the images. To accomplish this, we focused on leveraging the relative positions between two images, which allowed us to discard the sequential data dependence that was slowing down our implementation. Specifically, we obtained the relative x, y, z, and quaternion positions between two images at times t and t+1. We then stacked the left and right images for each of the two frames, resulting in a total of four images. This stacked image data was fed into a CNN, which learned the relative position and orientation between the two frames.

In figure 5 we can see our VO network architecture. VONet consists of a series of convolutional layers for feature extraction from the stacked image data. Specifically, we used a CNN with three convolutional layers, each followed by batch normalization, a ReLU activation function, and max-pooling. These layers were used to extract features from the input images.

The output of the convolutional layers was then flattened and fed into a fully connected layer for regression to x, y, and z coordinates, and quaternions. The architecture of this layer included two linear layers with ReLU activation functions. The final output layer consisted of seven nodes representing the predicted values for the position and orientation of the camera between the two image frames.

We trained VONet using the following hyperparameters:
Number of epochs: 20
Batch size: 128
Learning rate: 10e-5

$$L = w_p \cdot \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (z_i - \hat{z}_i)^2$$
$$+ \mathrm{w}_q \cdot \tfrac{1}{N}\textstyle\sum_{i=1}^{N}(q_{w_i} - \hat{q}_{w_i})^2 + (q_{x_i} - \hat{q}_{x_i})^2 + (q_{y_i} - \hat{q}_{y_i})^2$$
$$+ (\mathrm{q}_{z_i} - \hat{q}_{z_i})^2 \quad (1)$$

We used the mean squared error (MSE) loss function to optimize the network parameters during training.

By using this approach, we were able to significantly reduce the amount of compute and time required for processing the data, while maintaining the accuracy of the system. To evaluate the performance, we used L2 loss and compared the results to the ground truth. Through this process, we were able to develop a more efficient and effective deep learning-based visual-inertial odometry system that can be trained on limited data and provide fast inference for real-time navigation applications.

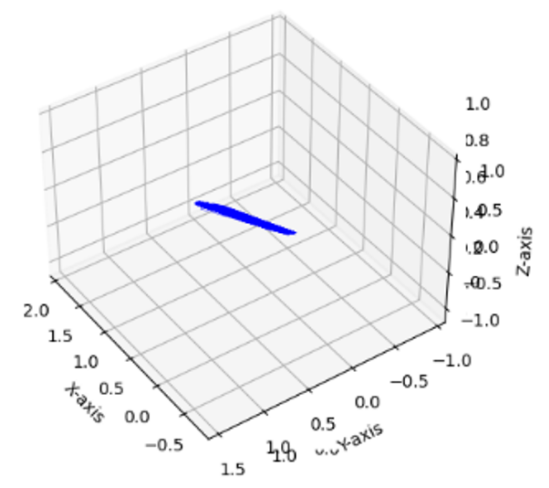


Fig. 7. VO Output

Next, we implemented Inertial Odometry (IO) network for our visual-inertial odometry system. As mentioned in the pre-processing part of our pipeline, we first matched the IMU timestamps with the ground truth timestamps. We used the angular velocities in the x, y, and z axis, and linear acceleration in the x, y, and z axis as the six input features for the network. Unlike the VO network, which predicts relative values, the IO network predicts absolute values for the camera's position and orientation.

The IO network consisted of a single LSTM layer and a fully connected layer for regression to the output values. The LSTM layer was used to capture temporal dependencies in the input data, while the fully connected layer was used to map the hidden state of the LSTM to the output values.

We trained the IO network using the following

hyperparameters:
Number of epochs: 100
Batch size: 32
Learning rate: 10e-2

We used the mean squared error (MSE) loss function to optimize the network parameters during training.

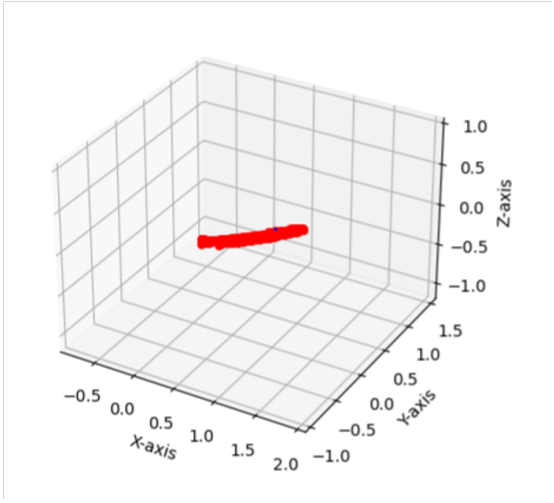


Fig. 8. IO Output

Finally, we implemented a CNN-based VIO network, but due to time constrains, we were not able to completely debug it. However, we have included it in our code for reference and are still trying to debug it to get final results.

E. Problems Faced

During the training of the VO network, we faced the issue of getting NaN values for the training loss after a certain number of epochs. This can be caused by various reasons, such as exploding gradients, vanishing gradients, or numerical instability. To address this issue, we tried several approaches.

First, we attempted to decrease the learning rate to prevent the optimizer from overshooting the optimal weights. However, this did not resolve the issue.

Next, we tried using Xavier weight initialization, which is a commonly used method for initializing weights in neural networks. This method helps to avoid the problem of vanishing or exploding gradients by keeping the variance of the activations constant across layers. However, this also did not help to resolve the issue.

We also attempted to use gradient clipping to prevent the gradients from becoming too large or too small. This can help to avoid the problem of exploding gradients, which can cause NaN values in the training loss. In addition, we added batch normalization to normalize the inputs to each layer and stabilize the learning process and in the end resolved the issue.

We faced a lot of issues in getting the plots from the .txt file. First of all our output values were quite small, thus it was difficult to visualize it on plot. Further, since the .txt had some blank values in it which threw a lot of errors and was difficult to debug. Finally we were able to get some plots from 3D Matplotlib. Also, since we assigned much higher weights for positions as compared that of the orientations, our plots came out to be almost linear, which was not good. Our idea behind doing so was that we need more accurate locations for the plot, rather than the orientation of the bot. Later, after discussing with Professor we realized it was not a good idea

to do it as the orientations were also important and were used to create the trajectory.

we implemented a CNN-based VIO network, but due to time constrains, we were not able to completely debug it. However, we have included it in our code for reference and are still trying to debug it to get final results.

REFERENCES

[1] Wang, S., Clark, R., Wen, H., Trigoni, N. (2017). DeepVO: Towards end-to-end visual odometry with deep Recurrent Convolutional Neural Networks. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) (pp. 2043-2050).

[2] Li, R., Wang, S. (2018). Deep visual-inertial odometry based on a 3D directional grid. arXiv preprint arXiv:1806.05632.

[3] Zhu, A. Z., Thananjeyan, B., Tompson, J., Goldberg, K. (2018). DeepTIO: A deep thermal-inertial odometry with visual hallucination. In Proceedings of Robotics: Science and Systems.

[4] Costante, G., Bellocchio, E., Valigi, P., Ricci, E. (2018). Exploring deep visual-inertial odometry on embedded systems. In Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 6099-6106).

[5] Choi, C., Dariush, B. (2018). DenseFusion: 6D object pose estimation by iterative dense fusion. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 3343-3352).

[6] Bloesch, M., Omari, S., Hutter, M., Siegwart, R. (2015). Robust visual inertial odometry using a direct EKF-based approach. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 298-304).

[7] Radwan, N., Valada, A., Burgard, W. (2018). VLocNet++: Deep multitask learning for semantic visual localization and odometry. IEEE Robotics and Automation Letters, 3(4), 4407-4414.

[8] Brossard, M., Barfoot, T. D., Peretroukhin, V. (2020). Deep-inertial-odometry: Learning latent representations for end-to-end visual-inertial odometry. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) (pp. 7554-7560).