POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Visual Odometry System for Drones Assisted by a Deep Neural Network

OLIVIER BROCHU DUFOUR

Département de génie mécanique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie mécanique

Mai 2021

Visual Odometry System for Drones Assisted by a Deep Neural Network

REMERCIEMENTS

Ce projet de maîtrise n’aurait pu voir le jour sans l’aide d’une multitude de personnes m’ayant épaulé de près ou de loin tout au long de mon parcours.

J’aimerais offrir des remerciements tout particuliers au professeur Sofiane Achiche, mon directeur de maîtrise, qui a su croire en moi et qui m’a donné la liberté créative nécessaire pour mener ce projet à terme. Merci au professeur Abolfazl Mohebbi, mon co-directeur, qui, grâce à ses commentaires et conseils, m’aura permis de réaliser un projet de qualité dont je suis fier. Ce fut un immense plaisir de réaliser cette maîtrise sous votre supervision.

Merci aux professeurs David Saussié et docteur Ali Safaei d’avoir accepté d’être membre du jury qui évaluera cette maîtrise.

Merci à mon cousin, Alexandre Duperré, qui m’a donné le courage d’entreprendre cette maîtrise et sans qui je n’aurais probablement pas survécu. Tes conseils, ton soutien et ton amitié m’ont permis de me rendre jusqu’au bout avec toute ma tête!

Merci à mes précieux parents, Christine et Francis, ainsi que mon frère, Jérémie, qui ont été autant là pour moi durant les moments difficiles qu’avec moi pour célébrer les victoires. C’est grâce à votre appui inconditionnel que ce projet a pu voir le jour.

Merci à mes amis, vous êtes une source inépuisable de délire et de divertissement, sans vous ce projet en temps de COVID aurait-été bien morose.

RÉSUMÉ

L'utilisation de drones devient de plus en plus courante pour de nombreux secteurs industriels. Toutefois, certaines limitations techniques, comme la navigation dans des environnements dépourvus de signal GPS, peuvent constituer un obstacle à leur adoption. La recherche dans le domaine de l'odométrie visuelle évolue rapidement et pourrait apporter des solutions à ce type de problèmes.

Les techniques actuelles d'odométrie visuelle sont généralement composées d’étapes permettant l'extraction et la recherche de points d’intérêts dans la séquence d’images ou la minimisation d’erreurs photométriques afin de reconstruire les matrices de transformation décrivant le mouvement du véhicule. Bien que populaires, les solutions actuelles peuvent souffrir d'une accumulation d'erreurs importantes et peuvent être coûteuses en termes de calcul. De nouvelles recherches utilisant des réseaux neuronaux profonds ont démontré des résultats prometteurs et pourraient éventuellement offrir des solutions aux lacunes existantes de ces techniques dites « géométriques ». Les techniques d'odométrie visuelle profonde actuelles combinent généralement des réseaux neuronaux convolutifs et des réseaux de neurones récurrents séquentiels afin de déduire l'odométrie visuelle d'une séquence vidéo donnée.

L'objectif de ce projet de recherche est de développer un système intelligent capable d'estimer l'odométrie visuelle d'un drone en temps réel. Le système intelligent utilise une nouvelle architecture neuronale profonde appelée SelfAttentionVO pour estimer, à partir d’une séquence vidéo, le mouvement d'une caméra fixée à un véhicule. Un utilitaire d'inférence capture le flux vidéo diffusé par le véhicule et utilise les prédictions de SelfAttentionVO faite sur les images de la vidéo pour assembler une estimation complète de la trajectoire. Bien que le système ne soit pas actuellement optimisé pour fonctionner sur un ordinateur embarqué, l'utilitaire d'inférence peut être utilisé sur un ordinateur de bureau ordinaire, offrant ainsi une alternative viable.

SelfAttentionVO a été entraîné sur les ensembles de données KITTI et Mid-Air en utilisant une fonction d’objectif basée sur l'erreur quadratique moyenne. L'architecture utilise un réseau neuronal convolutionnel, pour effectuer l'extraction de points d’intérêts dans l'image et utilise à la fois un réseau de type « long short-term memory » (LSTM) et un module d'attention multi-têtes pour modéliser les dépendances séquentielles de la vidéo.

Les résultats de tests montrent que SelfAttentionVO converge 48 % plus rapidement que DeepVO, un modèle similaire fréquemment utilisé à des fins de comparaisons dans la littérature. Par rapport à DeepVO, SelfAttentionVO offre également une réduction de 22 % de la déviation de trajectoire moyenne (KITTI Translation Error) et une réduction de 12 % de l'erreur absolue de trajectoire translationnelle (ATE) moyenne. Une étude ablative montre également que l'entraînement sur des données augmentées provenant de KITTI et de Mid-Air améliore considérablement les performances du modèle. Finalement, SelfAttentionVO est plus robuste aux perturbations visuelles que DeepVO.

ABSTRACT

Drone use is becoming more common in many industrial sectors, but technical limitations like navigation in GPS denied environments can be a barrier to adoption. Research in the field of visual odometry is rapidly evolving and could provide solutions to GPS-less drone navigation.

Current state-of-the-art visual odometry techniques use standard geometry-based pipelines involving image feature extraction and matching (or photometric error minimization) and transformation matrix estimation. Although popular, current solutions can suffer from significant drift, accumulation and some can be computationally expensive. New research using deep neural networks has shown promising performances and could eventually offer a solution to the shortcomings of existing geometry-based techniques. Deep visual odometry techniques combine convolutional neural networks and sequence modelling networks like recurrent neural networks (RNNs) to build an understanding of the scene and infer visual odometry from a given video sequence.

The objective of this research project is to develop an intelligent system capable of estimating the visual odometry of a drone in real time. The intelligent system uses a novel deep neural architecture called SelfAttentionVO to estimate, from consecutive video frames, the egomotion of a camera rigidly attached to a vehicle's body. An inference utility captures the live video feed from a drone and uses SelfAttentionVO's predictions on the video frames to assemble a complete trajectory estimation. Although the system is not currently optimized to run on an onboard computer, the ability of the Inference Utility to estimate real-time visual odometry from a video feed using a desktop computer provides a viable alternative.

SelfAttentionVO was trained on the KITTI and Mid-Air datasets using mean squared error loss. The architecture uses a convolutional neural network to perform image features extraction and couples a long short-term memory (LSTM) network to a multi-head attention module to model the sequential dependencies of the video.

Test results showed that SelfAttentionVO converges 48% faster than similar model DeepVO. Compared with DeepVO, it also revealed a reduction of 22% in mean translational drift (KITTI Translation Error) from the ground truth and an improvement of 12% in mean translational absolute trajectory error. An ablation study also showed that training on augmented data from KITTI and

Mid-Air significantly improved the performance of the model. In the end, SelfAttentionVO is more robust to noisy input compared with DeepVO.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ATE	Absolute Trajectory Error
BRNN	Bidirectional Recurrent Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
GPS	Global Positioning System
GPU	Graphics Processing Unit
HLS	HTTP Live Streaming
HPC	High-Performance Computing
IMU	Inertial Measurement Unit
LIDAR	Light Detection And Ranging
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
NED	North East Down
NLP	Natural Language Processing
RANSAC	Random Sample Consensus
RCNN	Recurrent Convolutional Neural Network
RGB	Red, Green, Blue
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
RTMP	Real Time Messaging Protocol
RTSP	Real Time Streaming Protocol
SLAM	Simultaneous Location And Mapping
SSD	Solid State Drive
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

1.1 Motivation

Drones, also called unmanned aerial vehicles or UAVs, are now omnipresent in many industrial sectors given to their flexibility and usefulness, even permeating popular culture as a tool of leisure. Even then, the mass adoption of these vehicles is still incomplete. Although shrinking, technical limitations still prevent their usage under certain conditions. For instance, it can be challenging to develop industry solutions for small, resource-limited drones that need to be operated in GPS-denied environments or that require extremely precise positioning [1][2]. Under normal circumstances, GPS is a crucial component of proper drone operation. It allows for precise and reliable location and odometry information. Current solutions to this problem are mainly based on technologies originally developed in the early 80s by NASA's rover development program. These algorithms use computer vision to calculate the movements of a camera, rigidly attached to the body of the vehicle, which are then assembled into complete trajectory translation and rotation estimates. This process is called visual odometry. Unfortunately, these algorithms, which rely on hard-coded geometry-based logic, are subject to error accumulation, and some are resource-intensive. New research attempts to tackle these issues using deep neural networks with surprising success. Some networks even reach state-of-the-art performances on popular datasets, but none of them are specifically designed according to the computational requirements of onboard computers. Currently, the vast majority of the best performing visual odometry algorithms, neural or otherwise, is either resource-inefficient or require extra sensors (inertial measurement unit or IMU, second camera, etc.). A small and computationally efficient single camera solution would be to visual odometry research what the panacea would be to pharmaceutical research.

1.2 Objectives

The objectives for this project have been formulated based on the limited requirements associated with small onboard computers, the current technological limitations and the desire to improve the state of visual odometry research. The main objective of this research is defined as follows:

To develop an intelligent system capable of estimating the visual odometry of a drone in real time.

To achieve this, the following sub-objectives will need to be met:

> **S.O. 1**: To design a supervised deep neural network capable of producing accurate visual odometry estimations.
>
> **S.O. 2**: To develop a program capable of feeding the live video stream to the deep neural network.
>
> **S.O. 3**: To characterize the performances of the intelligent system.

A series of hypotheses need to be validated to accomplish the objectives defined above.

Hypothesis 1: *Multi-head attention modules can increase the accuracy of visual odometry estimations.*

New research from the domain of natural language processing shows that neural networks using attention modules outperform other types of networks in sequence modelling tasks. Since visual odometry is fundamentally a sequence modelling problem, it is possible that a multi-head attention module can help increase the accuracy of visual odometry estimations.

Hypothesis 2: *Auxiliary task estimation can increase the accuracy of visual odometry estimations.*

Research in unsupervised learning for visual odometry has shown that auxiliary tasks help regularize the training and increase estimation accuracy. It is possible that auxiliary tasks can also help a supervised neural network.

Hypothesis 3: *Dataset expansion can increase the accuracy of visual odometry estimations.*

The benefits of dataset expansion for prediction accuracy are well documented in the literature. Yet, few papers implementing deep neural networks for visual odometry are actually using these techniques. It is unclear why this is the case, but, in all likelihood, dataset expansion techniques may have a positive impact on the accuracy of the network.

Hypothesis 4: *A deep neural network estimating the visual odometry of a drone can be run in real time.*

Although inference using a deep neural network is usually fast, it is unclear if it can run in real time. Validation or invalidation of this hypothesis is crucial to the accomplishment of S.O. 2.

1.3 Document Structure

The memoir is composed of six (6) chapters. Chapter 2 is the first chapter of the literature review and covers the current knowledge on relevant deep neural networks. The deep learning concepts presented in this chapter are mostly focused on computer vision and sequence processing, which are the main notions used in deep visual odometry estimation. Chapter 3 is the second and last chapter of the literature review. It focuses on the current state of knowledge of visual odometry research. It covers "geometric" visual odometry techniques, as well as supervised and unsupervised deep neural network-based techniques. Chapter 4 presents the tools and methodology used to test and validate the hypotheses and objectives. Chapter 5 focuses on and discusses the results of the research, which includes the architecture of the neural network, the benchmarked performances of the network and an ablation study to understand the contribution of the different architectural components and data augmentation techniques. Lastly, Chapter 6 provides an overview of the research project and offers avenues for future research.

CHAPTER 2 THEORETICAL NOTIONS

This portion of the literature review is meant to offer an overview of the research and notions involved in developing deep learning models.

2.1 Convolutional Neural Networks

Figure 2.1 depicts the general architecture of a multi-layer (2 in this case) feed-forward neural network. This type of network is considered to be one of the building blocks of modern deep neural network architectures. Every fully connected layer is composed of units that are all connected to every unit of its adjacent layers.

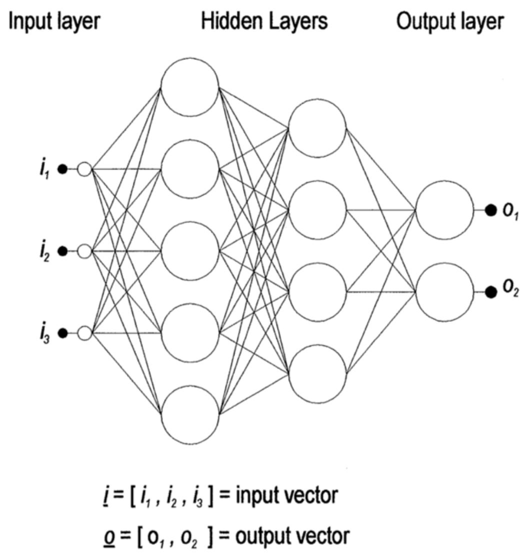

Input layer
Hidden Layers
Output layer
i_1
i_2
i_3
o_1
o_2
$\underline{i} = [i_1, i_2, i_3]$ = input vector
$\underline{o} = [o_1, o_2]$ = output vector

Figure 2.1 Multi-layer perceptron [3]

Mathematically, a feedforward layer can be expressed as

$$\underline{\boldsymbol{o}}(\underline{\boldsymbol{i}}) = \boldsymbol{\sigma}(\boldsymbol{W}^{\top}\underline{\boldsymbol{i}} + \boldsymbol{b}), \tag{2.1}$$

where $\underline{\boldsymbol{o}}$ is the output vector of the layer; $\boldsymbol{W}$ is a weight matrix; $\underline{\boldsymbol{i}}$ is an input vector; **b** is a bias vector; and $\boldsymbol{\sigma}$ is a non-linear activation function. These layer functions are then chained to obtain a multilayered feed-forward network. The numerical values of the weight matrix and the bias are computed using a gradient descent training algorithm [4] [3] to optimize the estimation accuracy of $\underline{\boldsymbol{o}}(\underline{\boldsymbol{i}})$.

Convolutional neural networks (CNNs) are a special form of the multi-layer feedforward neural network that employs matrix convolutions, often in its first few layers, to ingest grid-like data such as images. Like other types of neural networks, they can be used in classification, regression and density estimation tasks [4].

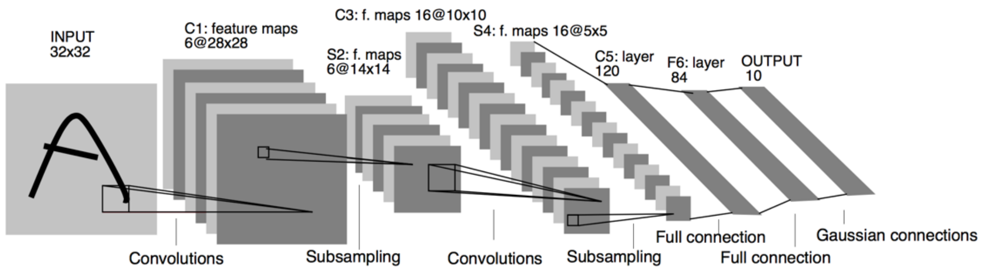


Figure 2.2 Architecture of the LeNet-5 [5]

A common basic convolutional neural network architecture is shown in Figure 2.2. As can be seen, convolutional layers are usually the first few layers. They reduce and filter the input to identify patterns. They are composed of three stages: the convolution itself, a non-linear activation function, and a pooling/subsampling layer, which reduces the dimensionality of the resulting vector. Lastly, CNNs will use fully connected feed-forward layers to generate a proper vectorized output [4].

When it comes to the analysis of large data inputs like images, traditional multilayered neural networks have a few shortcomings. In a more traditional network, to completely ingest a large input like an image, it is necessary to fully connect every pixel ($\boldsymbol{m} \times \boldsymbol{1}$) to every single neuron ($\boldsymbol{n} \times \boldsymbol{1}$) of the first layer. The resulting number of trainable parameters ($\boldsymbol{m} \times \boldsymbol{n}$) would just be too large. Moreover, the network would have a hard time generalizing images with similar content, but arranged differently since the input signal would be consumed as one monolithic block, leaving no room for the analysis of individual components within the image [4].

In order to efficiently ingest a complete large input like an image, one strategy is to process it piece by piece. In CNNs this is achieved by sequentially convolving pieces of the input with the layer's convolution kernel, effectively scanning the whole image all while reusing the same convolutional kernel values. This not only allows for the processing of potentially arbitrary large image inputs, but also significantly reduces the number of trainable parameters which makes the model smaller and more computationally efficient. This gives rise to three fundamental properties of convolutional neural networks: parameter sharing, sparse connections and equivariant representation.

Parameter sharing is the re-use of the same convolution kernel values for every image piece; it significantly reduces the memory footprint of the models.

Sparse connections happen because, unlike fully connected networks, only the input units (e.g.: pixels) convolved with the convolution kernel will affect an output unit. In contrast, in fully connected networks, all input units that are matrix multiplied with the weight matrix will affect every output unit. It is the reduced number of connections between the input and output units that make the model more computationally efficient.

Lastly, the equivariant property is a by-product of parameter sharing. Mathematical equivariance can be expressed as follows [4]:

$$\boldsymbol{f}(\boldsymbol{g}(\boldsymbol{x})) = \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x})), \tag{2.2}$$

where $\boldsymbol{f}$ is invariant to $\boldsymbol{g}$, so it does not matter if $\boldsymbol{g}$ is applied to the input or the output; the result will be the same. CNNs share the same property; they have translational invariance. For example, a feature identified in one portion of an image will also be identified if it is located elsewhere in the image. More concretely, consider the input units (pixels) $\boldsymbol{P}$, and a translational function $\boldsymbol{g}(\boldsymbol{x})$, which shifts the location of its input $\boldsymbol{x}$ by an arbitrary value $\boldsymbol{V}$. Then the translational invariance of a CNN can be expressed as:

$$\boldsymbol{conv}(\boldsymbol{g}(\boldsymbol{P})) = \boldsymbol{g}(\boldsymbol{conv}(\boldsymbol{P})) \tag{2.3}$$

Meaning that, given a feature that is translated by $\boldsymbol{g}$, when convolved with the convolution kernel, the same output units' value will be generated, but also translated by $\boldsymbol{g}$.

In a trained convolutional neural network, the convolutional layers can be seen as a series of filters that can extract “features” out of the image signal. As shown in Figure 2.3, in early layers, simple features like edges and corners can be identified in an image. As the signal progresses through the layers, those low-level image representations get combined and interpreted into higher-level representations like complex shapes [6].

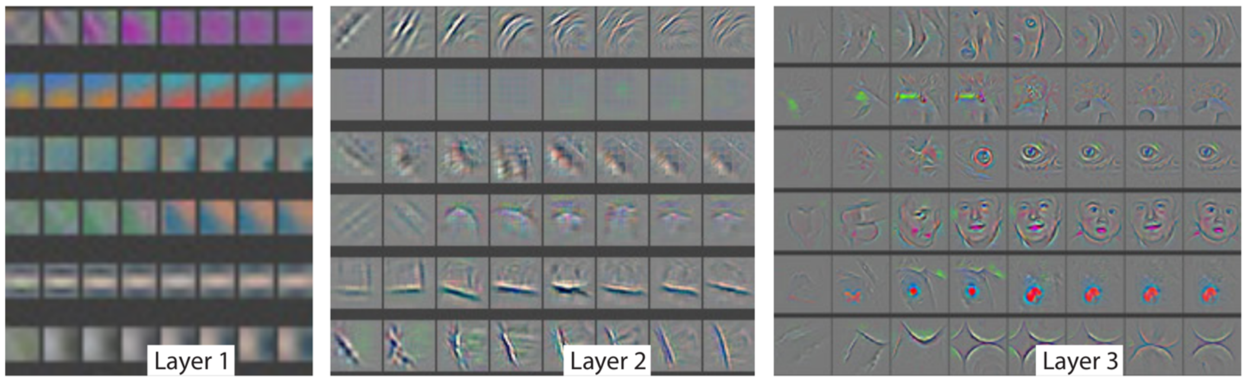


Figure 2.3 Visualized convolutional layers outputs [6]

As mentioned previously, convolutional layers are composed of three parts, the convolution step, the non-linearity step and the pooling step which we quickly glanced over. The main purpose of the pooling stage is twofold. It further reduces the dimensionality of the signal, which reduces the computational complexity of the network, and it introduces translational invariance to local translations of the input. Pooling is essentially a method for summarizing the values of the local neighbours of the output. “Max pooling” will return the maximum value and average pooling will return the average value of the neighbourhood. It is this high-level representation of the neighbourhood that introduces translation invariance because through this operation we lose some location information, making it less influential.

2.2 Recurrent Convolutional Neural Networks

Recurrent convolutional neural networks (RCNNs) are types of convolutional neural networks that have been designed to model grid-like data with sequential relationships, for example video. Representing sequential data with traditional neural networks can be challenging especially when presented with long sequences. In the scientific literature for visual odometry, representing sequential data with regular CNN is often achieved by stacking short snippets of sequential data and consuming it as one input [7], or feeding two adjacent frames simultaneously to two convolutional encoder networks that share weights [8][9]. Although these designs have been successful at modelling the relationship between adjacent time steps, they cannot model long-term dependencies.

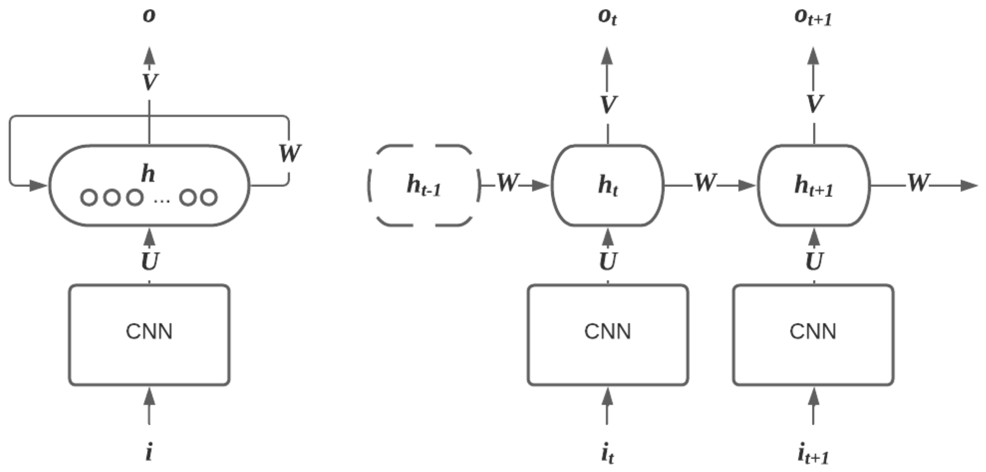


Figure 2.4 RCNN, folded and unfolded

Figure 2.4 shows a basic RCNN, design both folded and unfolded on many time steps. As can be seen, the first layers are dedicated to the convolutional neural network, followed by a recurrent neural network (RNN). We notice a weighted connection ($\boldsymbol{W}$) between the hidden layer $\boldsymbol{h}$ of the current time step and the previous time step. This connection addresses the issues associated with representing sequences using regular CNN. Called the recurrent connection, it feeds, as secondary input to the hidden layer, the network's previous hidden state. This allows the network to learn to represent the current time step in relation to the representation of the previous time step which is itself a summary representation of all of its previous time steps. This effectively allows the design to model whole sequences of data of arbitrary length.

Similar to the CNN, the RCNN employs parameter sharing in order to better generalize over the input data. However, RCNNs take this a step further by also sharing the RNN's parameter ($\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$) over each time step of the sequence. This is because having a recurrent connection is not enough to build a summary representation of all the time steps of the sequence. If different parameters were used every time, it would be impossible to build a model that has statistical strength across the time steps [4].

RCNNs are effective at modelling sequential data but are not well adapted to long sequences of data. Research has shown that learning using gradient descent on large sequences can be especially challenging [10][11]. This is because when training a model on long sequences, the gradient will tend toward zero. This is called the vanishing gradient problem. Zero or close to zero gradients mean that updates to the parameters of the network become so insignificant that the training becomes stuck. The gradients can also “explode”, reaching significantly large values, preventing the network from converging to a solution. The sensitivity to vanishing or exploding gradients in recurrent neural networks is caused by the fact that the weights of the network are recursively matrix multiplied with themselves. At time step $\boldsymbol{t}$, we can express this recursive relationship as follows:

$$\boldsymbol{h_t} \; = \; \boldsymbol{W}^{\top} * \boldsymbol{h_{(t-1)}} \tag{2.4}$$

Using the power method to find the eigendecomposition of $\boldsymbol{W}$, the recursive relationship can be simplified to:

$$\boldsymbol{h_t} \; = \; (\boldsymbol{W^t})^{\top} * \boldsymbol{h_0} \tag{2.5}$$

In this form, we can clearly see how the recursive relationship caused the weights to be raised to the power of $\boldsymbol{t}$. If the magnitudes of those weights are smaller than one, then the power method causes the values of $\boldsymbol{W^t}$ to descend toward zero as more time steps are computed. If they are greater than one, then they will grow exponentially. As the weights become smaller, so does the gradient computed during backpropagation, causing the updates made to the weights to become smaller as we propagate down the sequence. For more details about the decomposition from Equation 2.4 to 2.5, and the power method, please refer to chapters 10.7 and 8.2.5 of [4].

One of the most popular ways to address the vanishing gradient problem in recurrent neural networks is the addition of gated units to create gated RNNs. Gated units allow RNNs to preserve long-term information, which would otherwise be lost to vanishing gradients. When no longer useful, the gated units have a built-in mechanism to allow the network to forget about the accumulated information. These units are trained, along with the rest of the network, to automatically identify when it is appropriate to remember or forget long-term data representations.

2.2.1 Long Short-Term Memory (LSTM)

LSTM [12] is one of the most popular implementations of RNNs with gated units. LSTM recurrent networks have an architecture similar to regular RNNs. The difference is in their recurrent unit. Unlike an RNN unit, which concatenates the unit's previous output with the current input before passing it through a non-linear activation function, the LSTM's recurrent unit makes use of a second self-connection inside the LSTM's memory cell. This self-connection, which is linear, is where the cell's previous internal state will be read from, and where the current state will be written to. This internal recurrence allows for the cell's state and the gradient to flow across numerous time steps which is critical in preventing vanishing or exploding gradient. A weight on this connection allows the network to control how much of the previous cell's state to remember. A weight value of 0 means no information from the previous state gets saved; a value of 1 means that the full signal from the previous cell's state gets saved; and a value in between indicates that only a scaled-down signal gets saved. This weight is controlled by a trained forget gate, which can dynamically change the weight's value based on the current context (current input and previous output). The full architecture of an LSTM's memory cell is shown in Figure 2.5 and is also comprised of an input gate that adds the current cell state to the previous one and an output gate, which generates the cell's hidden state.

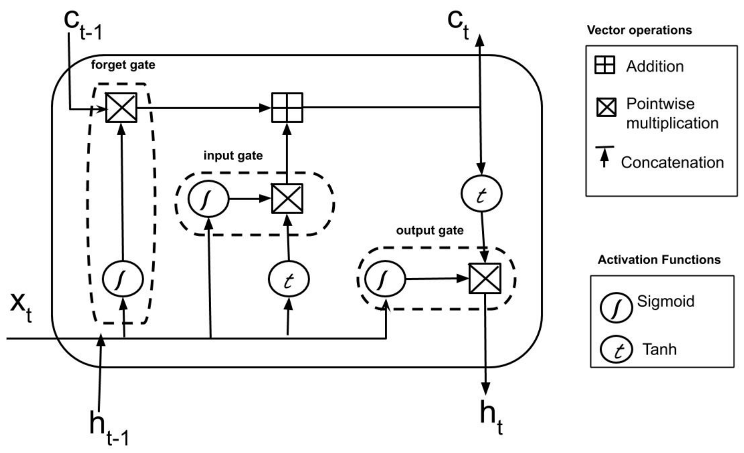


Figure 2.5 LSTM unit [13]

2.2.2 Bidirectional RNNs

It is possible to extend an RNN to not only model forward dependencies, but also backward dependencies using a bidirectional RNN (BRNN or Bi-RNN) [14]. Contrary to a regular RNN, a BRNN is trained simultaneously in the positive time direction and negative time direction. This allows for both the future state and the previous state to influence the current state. To achieve this, separate forward and backward recurrent units will respectively keep track of the forward and backward states. The general structure of a BRNN is shown in Figure 2.6.

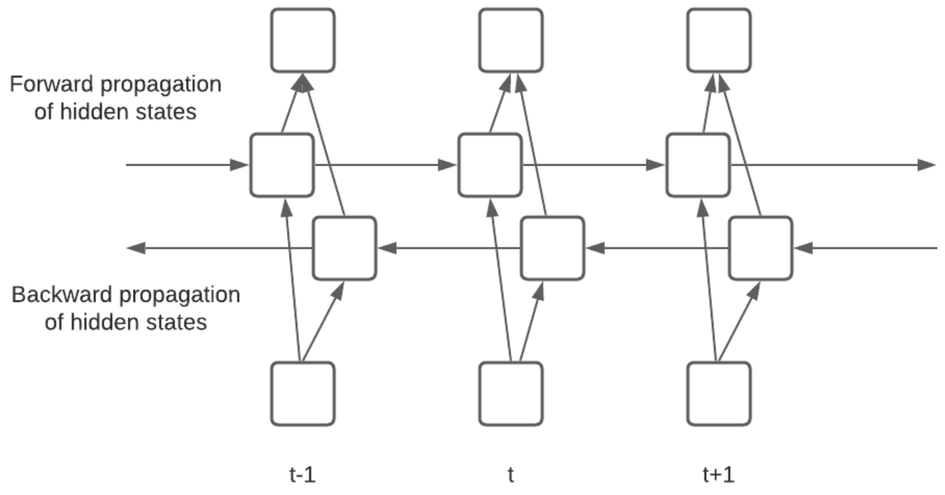

Forward propagation of hidden states
Backward propagation of hidden states
t-1
t
t+1

Figure 2.6 Structure of a BRNN

2.3 Attention Mechanisms in Deep Neural Networks

Attention mechanisms in deep neural networks have recently significantly gained in popularity for sequence processing applications such as NLP. The influential paper "*Attention is all you need*" by Vaswani et al. [15] demonstrated the effectiveness and computational efficiency of using an attention mechanism in Encoder-Decoder networks, called Transformers, in order to model sequential dependencies regardless of the length of the sequence.

Although there are different types of attention mechanisms, such as additive attention [16], Vaswani et al., introduce *Multi-Headed Scaled Dot-Product Attention*, which is used in the architecture presented in Chapter 5 for its computational efficiency.

Scaled Dot-Product Attention is defined using Equation 2.6:

$$\boldsymbol{Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V}, \tag{2.6}$$

where $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are matrices containing vector representations of the input sequence, also called $\boldsymbol{Queries}$, $\boldsymbol{Keys}$, and $\boldsymbol{Values}$ matrices, and $\boldsymbol{d_k}$ is the dimension of a $\boldsymbol{key}$ vector.

To model sequential dependencies using *Scaled Dot-Product Attention*, one must first compute vector representation of the input sequence's time steps $(\boldsymbol{t_0}, \dots, \boldsymbol{t_n})$ in order to produce matrices $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$. Often this is achieved through simple matrix multiplication of the input sequence's values with weight matrices $\boldsymbol{W^Q}$, $\boldsymbol{W^k}$ and $\boldsymbol{W^V}$. However, one can also use other methods like a bidirectional RNN [16] to create time-dependent vector encodings [17] of the sequence, which can then be concatenated into matrices $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$.

Once we have $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ we perform a simple dot product operation to obtain the cosine of the angle between the vectors of matrix $\boldsymbol{Q}$ and $\boldsymbol{K}$, and thus the similarity between the vectors. The resulting similarity matrix is then scaled. The authors of the paper believe "[…] that for large values of $\boldsymbol{d_k}$, the dot products grow large in magnitude, pushing the *softmax* function into regions where it has extremely small gradients" [15]. As discussed in Chapter 2, Section 2.2.1, small gradients could cause the training to "get stuck". Scaling mitigates this issue by preventing large dot product values. Once scaled, the *softmax* function is applied to the similarity matrix in order to obtain a probability of similarity the matrix. Lastly, the dot product of the probability matrix and matrix $\boldsymbol{V}$ is computed to get a matrix of values weighted according to the similarity between $\boldsymbol{Q}$ and $\boldsymbol{K}$. Figure 2.7 illustrates a toy example of the described procedure.

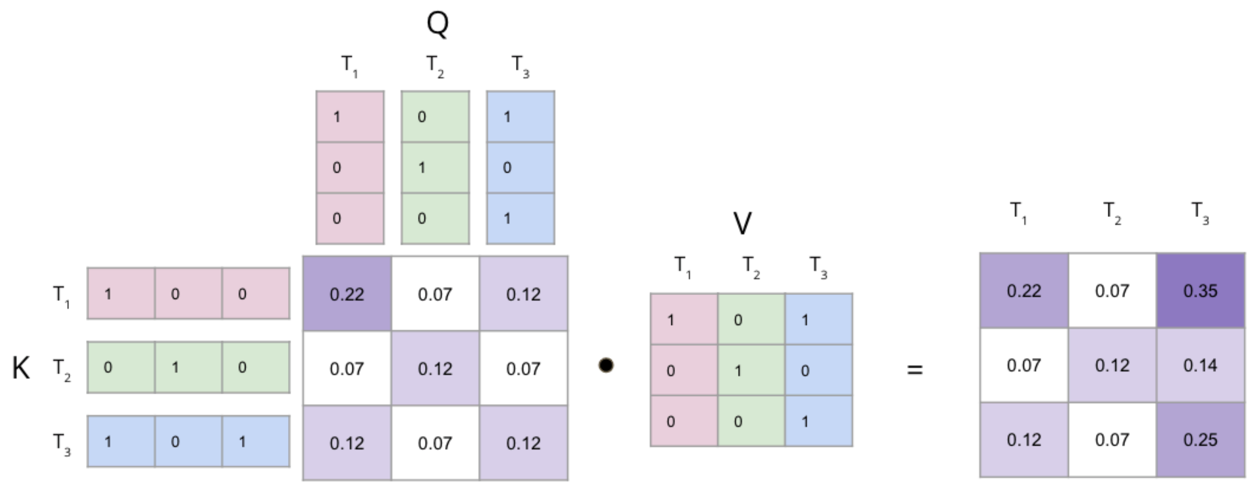


Figure 2.7 Illustrated toy example of the attention mechanism

This procedure is computed $\boldsymbol{h}$ times, in parallel, in a process called “multi-head attention”. As shown in Figure 2.8, in multi-head attention, attention is performed on $\boldsymbol{h}$ learned linear projections of matrices $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ (one for each head). Multi-head attention allows for each head to “attend to” information at different positions on the linearly projected matrix representations.

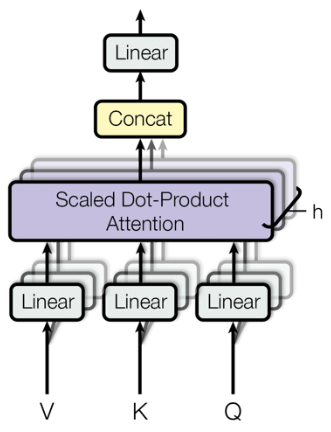


Figure 2.8 Multi-head attention [15]

When attention is performed on a single sequence of data (i.e., $\boldsymbol{K}$, $\boldsymbol{Q}$ and $\boldsymbol{V}$ are derived from the same source), and we are trying to model dependencies between its own elements, such as in language modelling, we talk about “Self-Attention”.

By favourably weighing sequential dependencies (similarities between the vectors of $\boldsymbol{K}$ and $\boldsymbol{Q}$), an attention mechanism creates a positive pressure for the network to learn to identify and track patterns in the signal throughout the sequence, regardless of its length.

CHAPTER 3 VISUAL ODOMETRY AND POSITIONING

This portion of the literature review is meant to offer an overview of the research and notions involved in developing visual odometry algorithms.

3.1 GEOMETRY-BASED METHODS

Visual odometry estimation using geometry-based methods were the first category of techniques developed in the 1980s to compute a vehicle's 3D motion in a scene, also called egomotion [18]. Research in this field was heavily driven by NASA's rover development programs and parts of the pipeline used by these techniques are still used in today's state-of-the-art methods [19]. Geometry-based methods can be used in monocular or stereoscopic algorithms. Stereoscopic algorithms are usually more accurate [20], since they can recover the scene's scale and 3D structure. However, they are reduced to a monocular problem if the only available features are distant [19].

3.1.1 Sparse Features Methods

Early research investigating sparse feature-based techniques can be traced back to the early 1980s with Moravec [21], who proposed one of the first computer programs to drive a robot through a cluttered environment using exclusively onboard cameras. Since then, research into visual odometry using sparse features methods has flourished and has heavily focused on improving the accuracy of those systems through schemes such as error modelling [22], pose refinement [23][24] and outlier rejection [25], to name a few.

Used in both monocular and binocular set-ups, sparse features methods usually adhere to the following algorithmic structure to achieve visual odometry.

First, salient image features are detected and extracted from input frames using common techniques such as Difference of Gaussians, to name one. The sparse features algorithm then attempts to describe the feature using a vector summarizing the appearance of the feature. This facilitates the matching step. During this step, the algorithm attempts to match features identified in the new frame to the previously seen ones in order to construct motion vectors. SIFT [26] and SURF [27] are common examples of algorithms used to perform feature extraction and matching. At this point, it is not uncommon to perform outlier detection and removal. Often this is achieved by removing features that cannot be properly matched between frames within an error margin. One such popular methods is called RANSAC (Random Sample Consensus) Refinement, which consists of eliminating features whose transformations (rotation, translation) do not match (within a specified threshold) the estimated transformation of a randomly selected subset of features [28].

Once the features are extracted and matched, the motion of the camera between two frames ($\boldsymbol{I_{k-1}}, \boldsymbol{I_{k}}$) is estimated by computing a transformation matrix $\boldsymbol{T_k}$ from the features sets $\boldsymbol{f_{k-1}}, \boldsymbol{f_k}$ (where $\boldsymbol{k}$ is the current time step). This is done by first estimating, using [28], the essential matrix $\boldsymbol{E_k}$, which describes the geometric relationship bound by the epipolar constraint (Equation 3.1) of an image pair [19]

$$\boldsymbol{\tilde{p}'^{\mathrm{T}} E \tilde{p} = 0}, \tag{3.1}$$

where $\boldsymbol{\tilde{p}'}$ and $\boldsymbol{\tilde{p}}$ are corresponding feature points belonging to $\boldsymbol{f_{k-1}}, \boldsymbol{f_k}$ respectively, that lies on the epipolar line as illustrated in Figure 3.1.

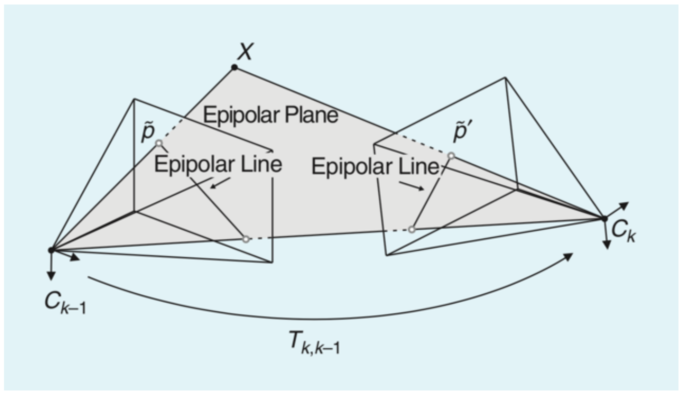


Figure 3.1 Epipolar constraints [19]

Lastly, the essential matrix is broken down into the proper transformation matrix using the technique described in [28].

To reduce the accumulated drift, the poses will be refined through loop closure, a location recognition algorithm, or bundle adjustments, which consist of minimizing the reprojection error in order to obtain the "[…] 3D structure and viewing parameter (camera pose and/or calibration) estimates" [29]. The issue of drift is further discussed in Chapter 3.3.

3.1.2 Direct Methods

Unlike feature-based methods, direct methods skip the feature extraction and matching steps of the pipeline entirely and instead process the whole image directly in the motion estimation step. Direct methods do this by minimizing an error metric that uses information from "[…] all the pixels in the image (such as brightness, or brightness-based cross-correlation, etc.)" [30]. Common methods like DTAM (Dense Tracking and Mapping) [31] and LSD-SLAM (Large-Scale Direct Simultaneous Localization and Mapping) [32] minimize the photometric error of each pixel between consecutive images in order to obtain the rigid body transformation of the camera. Although there is a wide variety of error metrics, most rely on the brightness consistency constraint and a "global motion model" [33]. The brightness consistency constraint states that an image $\boldsymbol{I}$ at time $\boldsymbol{t}$ and location $\boldsymbol{x},\boldsymbol{y}$, should be the same at time $\boldsymbol{t+1}$ given a small displacement $\boldsymbol{x+u}$, and $\boldsymbol{y+v}$ [30][34][35]. This constraint can be formally written like this:

$$\boldsymbol{I(x,y) = I(x+u, y+v)} \tag{3.2}$$

Global motion models are a family of algorithms that are used to further constraint the overall motion estimate by approximating the camera motion induced in the image. Rigid body motion is an example of a global motion model [33].

Like sparse features methods, the resulting estimated pose can be refined through different techniques. For example, during the tracking process, DTAM and LSD-SLAM will build a depth map of the scene and use it to reduce the uncertainty of the pose estimation. Note that because of their ability to build a map of the scene, DTAM and LSD-SLAM are not strictly visual odometry algorithms, but rather SLAM (Simultaneous Localization and Mapping) algorithms.

3.1.3 Direct vs. Feature-Based Methods

Both direct and feature-based visual odometry techniques are considered state of the art and choosing between them really boils down to the specific requirements of the problem at hand.

Feature-based techniques really shine in highly textured scenes since it is easier to extract robust features. Given these algorithms only extract a few hundred points from an image, their tracking step is usually fast. However, the extraction of those data points can be computationally expensive depending on the scene and the algorithm. Feature-based methods are also very robust when dealing with dynamic elements in a scene, such as pedestrians, due to their outlier rejection mechanism. However, their performance significantly drops when confronted with low-texture scenes since it becomes difficult to extract reliable features from it [36].

Unlike sparse features methods, direct methods are more robust when presented with textureless scenes because they ingest the complete image and do not perform any feature extractions, opting instead for minimizing the photometric error between consecutive frames. Moreover, with the extent of the input data used, they can produce dense maps of the scene, which is especially useful in SLAM systems. However, because they do not carry out feature extraction, description and matching, most of the computing effort is spent tracking and mapping the scene — a process more computationally expensive than feature-based methods. Lastly, they are very sensitive to illumination changes, because these methods assume consistent lighting intensity, a factor that can be hard to control in real-world use cases. [36]

3.1.4 Performance Geometry-Based Methods

Table 3.1 offers a summary of the best ranking geometry-based visual odometry methods and their performance on the KITTI dataset, which is a popular dataset used to train and evaluate visual odometry and SLAM algorithms. Only the top published geometry-based algorithms implementing a complete visual odometry pipeline that does not use LIDAR (Light Detection and Ranging) were included.

Table 3.1 KITTI Leaderboard summary of published geometric methods [20]

Rank	Algorithm	Method	Stereo/Mono	Translation error	Rotation error
9	SOFT-SLAM [37]	Feature	Stereo	0.65%	0.0014 [deg/m]
22	RADVO	Feature	Stereo	0.82%	0.0018 [deg/m]
29	DGVO	Direct	Stereo	0.86%	0.0031 [deg/m]
40	Stereo DSO	Direct	Stereo	0.93%	0.0020 [deg/m]
102	FTMVO	Feature	Mono	2.24%	0.0049 [deg/m]
103	PbT-M1	Feature	Mono	2.38%	0.0053 [deg/m]

The compiled benchmark data show that algorithms using stereo vision offer a much greater accuracy compared with their monocular counterpart. This is because stereo systems can have a better 3D understanding of the scene, and can therefore much more easily recover its scale, making them less susceptible to the accumulation of scale drift over time, which leads to inaccuracy [19][38][39]. Unfortunately, even if cameras are now more affordable, not all systems are equipped with stereoscopic vision systems and have the computational power to do so. This is why improvement in monocular vision systems, which are more common and usually have lower computational costs, are still of interest to the scientific community.

3.2 Deep Learning-Based Methods

As discussed earlier, standard monocular odometry techniques underperform compared with stereoscopic techniques [20]. This is attributable to their low understanding of the three-dimensional geometry of the scene, which leads to significant scale errors and drift. However, there is still significant interest in overcoming these hurdles, and researchers have turned to deep neural networks as a potential solution to monocular odometry estimation.

In the literature, there are two main approaches to deep visual odometry: supervised and unsupervised. Supervised techniques learn to estimate the probability distribution $\boldsymbol{p(y|x)}$ that best describes a training set of data composed of inputs $\boldsymbol{x}$ and their associated output $\boldsymbol{y}$. This is done by minimizing a cost function that penalizes the distance between an estimated output $\boldsymbol{y'}$ and the actual output $\boldsymbol{y}$ through a gradient descent algorithm [4]. However, unsupervised visual odometry techniques learn to extract a probability distribution $\boldsymbol{p(y|x)}$ best describing the egomotion of a camera from a set of unlabelled training data (i.e., data only composed of inputs $\boldsymbol{x}$ without any associated output $\boldsymbol{y}$). For visual odometry, this is mostly done by minimizing a cost function, through gradient descent, which penalizes the lack of consistency between two or more different but related tasks. This supervisory signal forces the network to learn to build a three-dimensional understanding of the scene and its motion, represented in the unlabelled dataset.

Although unsupervised deep visual odometry techniques currently outperform supervised techniques on the KITTI dataset [20], they result in larger networks that are not necessarily best suited for power- and memory-restricted vehicles like drones.

3.2.1 Supervised Networks

Early work by Konda et al. [7] investigated the use of convolutional neural network for visual odometry by attempting to extract the change in velocity and direction from the estimated depth of a stereoscopic sequence. Unfortunately, it characterized the problem as a classification one instead of a regression one, which limited the performances of the network. In later work, DeepVO by Wang et al. [40] further explored the use of CNN for visual odometry by coupling it with an LSTM network in order to extract sequential dependencies and infer pose from raw images. Apart from the novel addition of the LSTM network, DeepVO also introduces the use of FlowNet's [41] convolutional architecture and weights for the convolutional layers. When trained, the convolutional layers learn to extract features that are similar to the optical flow features extracted by FlowNet. This influential paper reports competitive results to state-of-the-art techniques. MagicVO by Jiao et al. [42] builds on DeepVO's architecture, but replaces the LSTM by a bidirectional LSTM to allow the network to learn information not only from the previous time steps, but also future ones.

Auxiliary task learning as a means to re-enforce consistency between predicted outputs is a very common strategy in the literature. Valada et al. (VLocNet) [8] showed how the problem of monocular visual odometry could be formulated as an auxiliary task to global pose regression. The proposed architecture uses a CNN network to predict odometry, which shares parameters with a second network used to predict global pose. The paper also uses a loss that enforces geometric consistency to jointly train both networks. The follow-up paper describing VLocNet++ by Radwan et al. [9] expands this idea of auxiliary learning by introducing a large neural network, achieving state-of-the-art performances, that is jointly optimized to predict visual odometry, global pose estimation and semantic segmentation. Interestingly, not only does the network outputs semantic segmentation, but it also uses it to further refine its pose estimation by focusing its "attention toward more informative regions of the scene" [9]. Lin et al. [43] explored auxiliary task learning by jointly training two RCNNs to predict both global and relative pose, using a loss that enforces temporal geometric consistency between the two predictions. In Parisotto et al. [44], the authors proposed an architecture inspired by DeepVO, but replaced the traditional RNN with temporal convolutions. The estimated relative poses are assembled into global poses, which are then refined by a "neural graph optimizer" composed of a series of multi-head attention modules [15] and temporal convolutions.

3.2.2 Unsupervised Networks

The most often stated motivations for using unsupervised learning are the lack of access to properly labelled datasets and the costs associated with producing them. Not only are unsupervised networks a great solution to these problems, but they also offer very competitive performances compared with state-of-the-art methods [20].

SfMLearner by Zhou et al. [45] introduced the use of view synthesis as a supervisory signal to learn both depth and pose in an unsupervised fashion. The idea builds on an earlier paper [46] that demonstrates that new points of view of a scene, which are synthetically generated, can be used as a metric to evaluate the quality of optical flow and stereo correspondence estimations. The architecture of SfMLearner forces the network to generate depth and pose data, which are used to reconstruct an image from the scene from a different position. In order to properly minimize the error of view synthesis generation, it will have to properly learn the scene's geometry and camera motion. Although it shows competitive results, the performances of the network are strongly affected by moving objects, occlusions, and non-Lambertian surfaces. UnDeepVO by Li et al. [47], built on the ideas of SfMLearner and introduced a very similar network with a different training pipeline. Notably, UnDeepVO trains on a stereoscopic image pair, using both images to compute *Spatial Image Losses* in the form of a left-right *photometric consistency loss* and a left-right *pose consistency loss*. They also use the right images to compute the *Temporal Image Losses* using the *photometric consistency loss* of consecutive monocular images and the *3D geometric registration loss* of consecutive monocular images. Interestingly UnDeepVO only trains on stereoscopic images; therefore, inferences are made on monocular images making it, like SfMLearner, a monocular network. Unlike other monocular networks, UnDeepVO can approximate scale, which makes it very accurate. Like SfMLearner and UnDeepVO, Li et al. [48] used depth as an auxiliary task and a composite *photometric consistency loss* to predict pose. However, unlike the previously mentioned papers, loop closure was also used to optimize pose prediction. Contrary to SfMLearner and UnDeepVO, Lyer et al. [49] introduced a lighter network that did not need to learn depth as an auxiliary task in order facilitate learning. Instead, the network uses an RCNN architecture, often seen in supervised networks, for training using the *Composite Transformation Constraints loss*. The loss forces the network to produce pose estimations that are geometrically consistent and follow the law of composition of rigid body transformations. Unfortunately, the network requires an extra pre-training step, and generalization to unseen data is poor. D3VO by Yang et al. [50]

jointly learned to estimate depth, relative pose and the photometric uncertainty between input images, by minimizing the photometric error through a composite loss. By predicting the photometric uncertainty, the network “down-weights” pixels that violate the brightness constancy assumption [51]. This novel proposal allows the network to outperform other unsupervised monocular deep visual odometry networks on the official KITTI visual odometry leaderboard [20], which is regularly updated with new techniques.

3.3 Drift

Since calculating visual odometry for a complete sequence is an incremental task, small errors will accumulate over time leading to significant discrepancies between the predicted motion and the ground truth [19]. For many applications, keeping this drift as small as possible is critical and is at the heart of odometry research.

Monocular visual odometry is especially sensitive to drift since not only is the 6 degrees of freedom of the camera unknown, but so is the scale of the scene (the 7th degree of freedom). This is because, unlike stereo systems, there is very little three-dimensional information in a monocular frame that can help connect distant portions of a scene together, while keeping the scale of each scene portions coherent with one another [39]. The farther away scene features are, the less their motion is going to have a positive effect on pose estimation. [24]

Loop closure is the process of correcting trajectory estimates by identifying the intersections in a path. Loop closure algorithms keep a dictionary of previously seen location estimations. When an intersection is detected it recursively corrects the trajectory estimates leading up to it so that the current location is close to the location stored in the dictionary. This effectively reduces the drift of the trajectory [19].

Other strategies in reducing drift can also involve bundle adjustment as mentioned in Chapter 3, Section 3.1.1, or completely eliminating (or significantly reducing) the uncertainty of one or more degrees of freedom. For example, adding gyroscopes to reduce orientation uncertainty, or adding a second camera to eliminate scale uncertainty.

CHAPTER 4 MATERIALS AND METHOD

4.1 Problem statement

Given a camera rigidly attached to the body of a drone and sharing the same coordinate frame, then the rigid body transformation of the camera from time step $\boldsymbol{i-1}$ to time step $\boldsymbol{i}$ can be expressed as a function of image inputs $\boldsymbol{I_{i-1}}$and $\boldsymbol{I_i}$:

$$\boldsymbol{T_{i-1,i}} = \boldsymbol{f(I_{i-1}, I_i)}, \tag{4.1}$$

where $\boldsymbol{T_{i-1,i}} \in \mathbb{R}^{\boldsymbol{4\times4}}$ is a transformation matrix, which can be written as:

$$\boldsymbol{T_{i-1,i}} = \begin{bmatrix} \boldsymbol{R_{i-1,i}} & \boldsymbol{t_{i-1,i}} \\ \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \tag{4.2}$$

$\boldsymbol{R_{i-1,i}} \in \boldsymbol{SO(3)}$ is a rotation matrix and $\boldsymbol{t_{i-1,i}} \in \mathbb{R}^{3\times\boldsymbol{1}}$ is a translation vector [19]. Since transformation matrix $\boldsymbol{T_{i-1,i}}$ can be composed with the previous transformation in order to obtain pose at time step $\boldsymbol{i}$:

$$\boldsymbol{P_i} = \boldsymbol{P_{i-1}T_{i-1,i}} \tag{4.3}$$

where $\boldsymbol{P_i} \in \mathbb{R}^{\boldsymbol{4\times4}}$ is a transformation matrix representing the position and orientation of the vehicle at time step $\boldsymbol{i}$ [19].

Then the set of all poses from time step $\boldsymbol{0}$ to $\boldsymbol{n-1}$, is the complete trajectory of the vehicle computed from the sequence of images $\boldsymbol{I_{0\to n-1}}$ [19]

$$\boldsymbol{P_{0\to n-1}} = \{\boldsymbol{P_0}, \dots, \boldsymbol{P_{n-1}}\} \tag{4.4}$$

This project aims to build a neural network that can approximate $\boldsymbol{f(I_{i-1}, I_i)}$ by learning the probability distribution $\boldsymbol{p(T_{i-1,i}|I_{i-1}, I_i)}$ (Chapter 3, Section 3.2) that best describes the training data.

4.2 Datasets

Training an efficient deep neural network can be a challenging task. Careful selection and preparation of the datasets are critical to ensure that the network can learn to regress Equation 4.1. Table 4.1 outlines the set of requirements used to select the datasets.

Table 4.1 Dataset selection requirements

Dataset selection requirements
Has sequences of images captured from an RGB camera rigidly attached to a moving vehicle
The vehicle of one or more datasets must be a drone
Offers six (6) Degrees of Freedom ground truth pose data for the vehicle or the camera
Is large enough to be broken down into a training, validation and testing dataset
Is well documented

When selecting more than one dataset, we must also assume that there exists a function describing the relationship established in Equation 4.1, which can be used on all the datasets. In order to estimate a function that follows the previous assumption, and because deep neural networks tend to store their training data as a superposition in their learned parameters [52], it is crucial that the selected datasets offer a wide variety of scenes. Using these requirements, the KITTI [53] dataset and the Mid-Air [54] dataset were selected.

4.2.1 KITTI Dataset

The KITTI [53] dataset is arguably the most used dataset in visual odometry research. It was selected because it satisfies the outlined requirements, and because its wide use facilitates the process of benchmarking of the developed model against the techniques found in the literature.

The dataset offers 11 video sequences with ground truth pose data. The video sequences are of a car driving in a small city. The ground truth pose data (3 × 4 transformation matrix) are supplied by a GPS/IMU sensor (OXTS RT 3003 [55]) with an open sky error of less than 5 cm. The ground truth pose data is projected in the left camera frame, which has the x-axis pointing right, the y-axis pointing up and the z-axis pointing forward. For the purpose of this project, the video sequences from the left PointGrey Flea2 colour camera was used. The camera sensor runs at 10 Hz and the final image resolution is 1328 × 512. Figure 4.1 shows the sensor set-up used for KITTI data collection. The yellow rectangle shows the location of the left camera. It also shows how the GPS and IMU's coordinate frame are transformed to the camera's coordinate system.

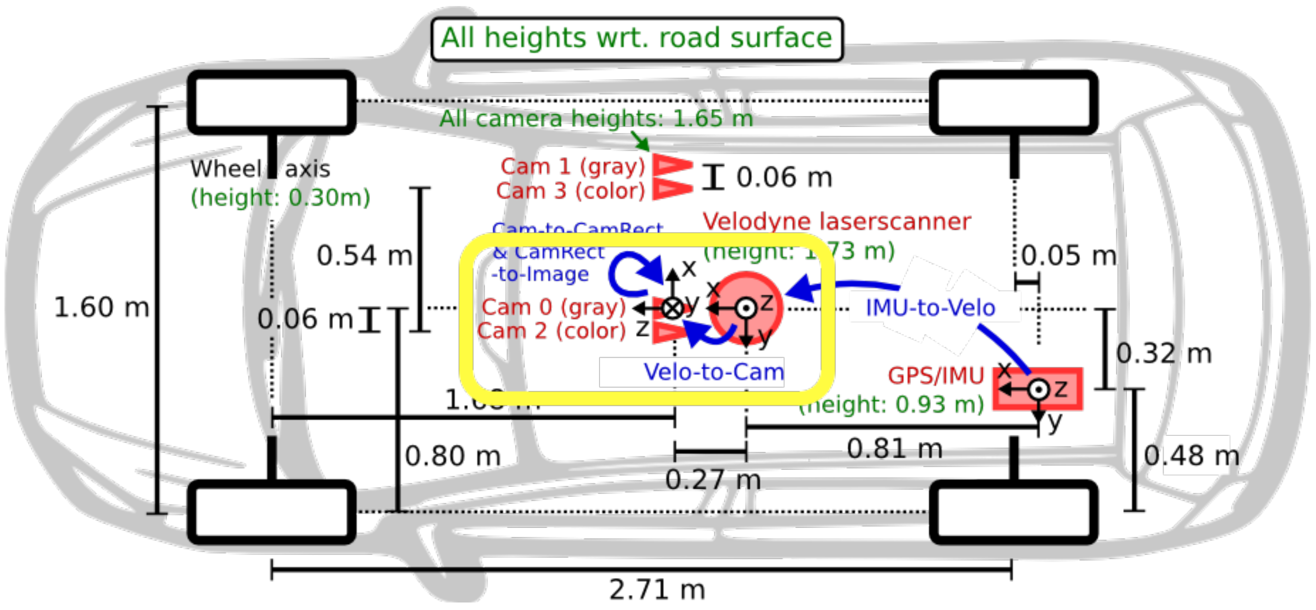


Figure 4.1 KITTI sensor set-up [56]

The dataset was segmented so that sequence 00, 01, 02, 05, 08 and 09 are used as training data, sequence 03, 04 and 06 as validation data and sequence 07, 10 as test data.

4.2.2 Mid-Air Dataset

The Mid-Air [54] dataset offers photo-real simulated aerial drone footage and ground truth pose data. The trajectories cover a variety of scenes including different seasons and climates. Mid-Air was selected because of the large number of video sequences available, the accuracy of the ground truth data and the stunning visual quality of the scenes, which increases the likeliness that a trained model will generalize well to real aerial video input. Moreover, it satisfies the outlined requirements.

The dataset offers 30 training trajectories in four (4) different weather conditions (sunny, sunset, foggy, cloudy) and 24 training trajectories in three (3) different seasons for a total of 216 training trajectories. It has five (5) validation trajectories in four (4) weather conditions, six (6) validation trajectories in three (3) different seasons for a total of 38 validation trajectories. Lastly, the dataset has three (3) test trajectories in three (3) climate conditions (foggy, sunny, sunset) for a total of nine (9) test trajectories for the purpose of benchmarking. All trajectories were used during their respective phases (training, validation, testing).

Mid-Air's virtual sensors (GPS, IMU, left camera, etc.) are rigidly attached to the body of the vehicle and aligned at the body frame's origin. The body frame and the world frame are aligned at time step 0, with the drone's yaw equal to 0. The body and world frame use the North, East, Down convention (x forward, y left, z down). Figure 4.2 shows Mid-Air's sensor set-up. In this figure the blue cube represents the IMU and GPS sensors.

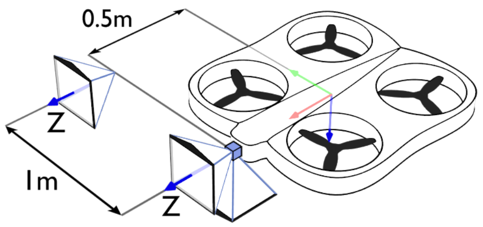


Figure 4.2 Mid-Air sensor set-up [54]

Each trajectory's data is organized in a highly efficient HDF5 [57] database file, which contains the ground truths and the path to the training data. The ground truth position and orientation are sampled at 100 Hz. The former is stored as a 3×1 translation vector (metres), and the latter is stored as a quaternion (radians), where the real component of the quaternion (also called the scalar) is listed first. The images used in this project are the ones provided by the left RGB camera which sampled the scenes at 25 Hz and at a resolution of 1024×1024 pixels.

4.3 Training Environment

For this project, a complete training environment was developed. It was engineered to be modular and allow for the rapid development and testing of deep neural network architectures, and loss functions. It uses PyTorch 1.6.0 [58], a popular machine learning library, for all its machine learning-related tasks.

4.3.1 Environment Architecture

Figure 4.3 offers a simplified overview of the training environment. This UML package-class diagram [59] shows the organization of the system's class relationships. Most class abstractions presented in this diagram have implementations, which are not shown for the sake of simplicity.

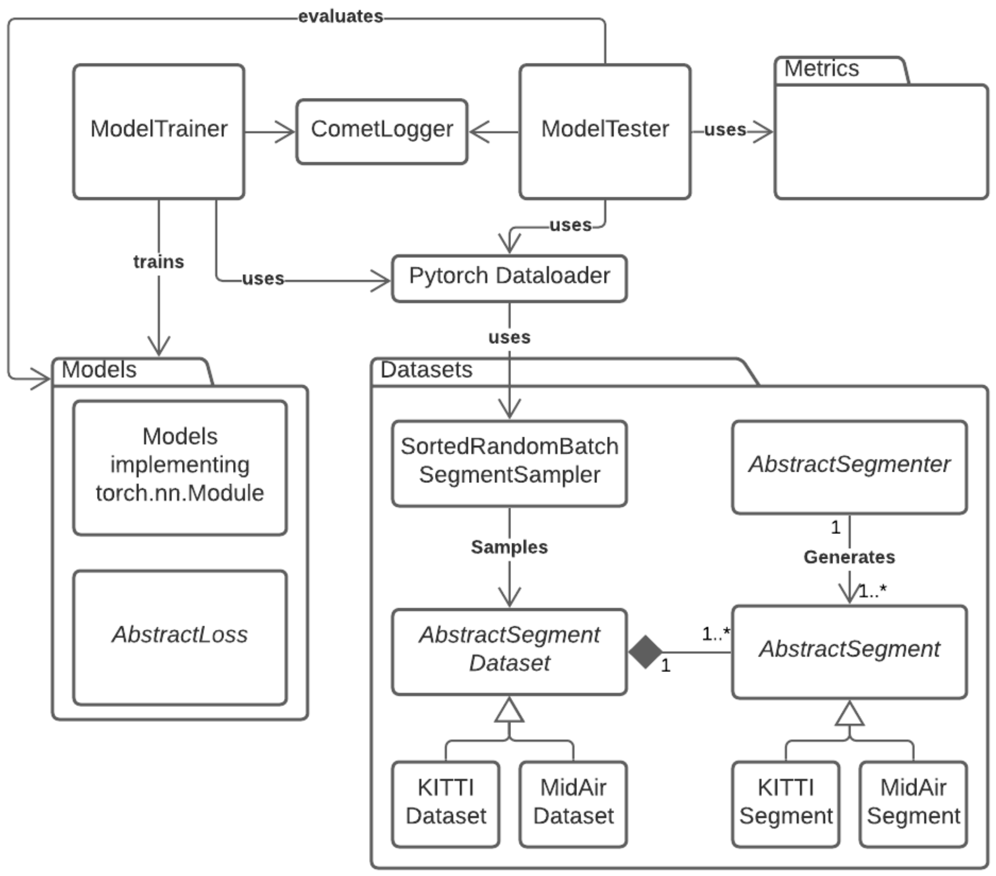


Figure 4.3 Training environment Package-Class diagram

As the name suggests, class *ModelTrainer* oversees training the deep neural models. It acquires training segments through a PyTorch *Dataloader* and feeds them to the model during the forward pass. It computes the loss using an implementation of *AbstractLoss* and updates the model's weights using backpropagation.

ModelTester evaluates the model's performances using the metrics located in the *Metrics* package. It also loads test trajectory segments using a Pytorch Dataloader.

Both the *ModelTrainer* and the *ModelTester* log their progress using the *CometLogger* static class, which offers an interface to Comet.ML, a cloud-based logging platform.

PyTorch Dataloader manages access to the data. It uses *SortedRandomBatchSegmentSampler* to randomly sample the dataset for batches of trajectory segments.

Datasets implement *AbstractSegmentDataset.* This class manages access to the segments of all the dataset's trajectories. It ensures that the returned segments are appropriate for the model. For instance, it can transform the image and pose data (resize the images, transform the coordinate system, etc.) to fit the model's requirements and system configurations.

Implementations of *AbstractSegmenter* are used to segment dataset trajectories into short segments of a couple of frames, these segments are represented by implementations of the *AbstractSegment* class, which manages and transforms the raw data so that the segment's initial world and camera frame are aligned at the origin.

It is required that every dataset implement the abstract classes of the *Datasets* package as shown in Figure 4.3. This means that, in the current codebase, there is an implementation for the Mid-Air and KITTI datasets.

4.3.2 Automated Training Pipeline

In order to accelerate model training and testing, an automated training pipeline was set up to run training sessions automatically on Compute Canada's Cedar cluster when the project's code is updated. The full pipeline is deployed across a variety of cloud services as shown in Figure 4.4.

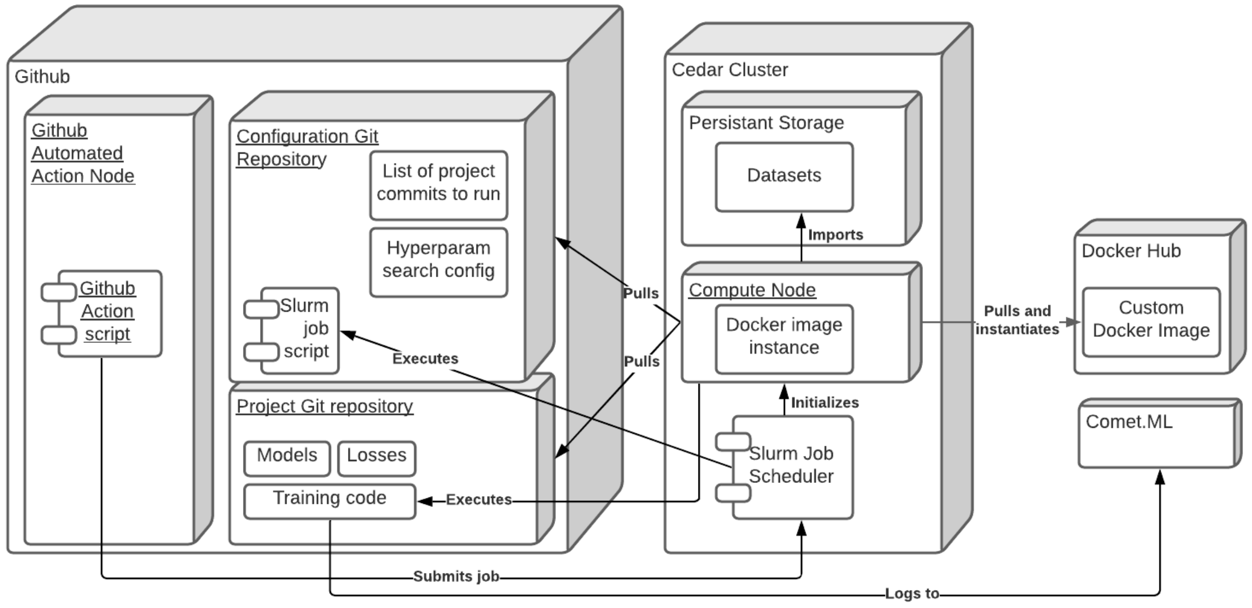


Figure 4.4 UML Deployment Diagram [59] of the Automated Training Pipeline

In this pipeline, Github is used as both a code repository and as an entry point for the automation. It hosts two git repositories. The first one stores the project's main code, which includes the models and training code. The second one is a configuration repository containing a list of the main project commits, a configuration file for possible hyperparameter search, and the automated job script. Github also hosts an automated *Action* node, which is instantiated when *git push* operations are performed on the configuration repository.

As mentioned earlier, Compute Canada is used to run the training. Compute Canada [60] is a network of supercomputers freely available for academic research. It is composed of six compute clusters located all over the country and designed to meet a varying set of computational requirements. For this project, the Cedar cluster [61] was selected because it offers general-purpose GPU compute nodes equipped with up to four (4) Nvidia V100 Volta with 32 Gb HBM2 memory, 800 Gb SSD disk storage and Internet access (required for logging). These powerful nodes are well suited for computer vision tasks since they can easily store and process large image datasets.

To run the training, the Cedar cluster use three main components. The persistent storage is used to keep an archived copy of the datasets; it resides outside of the short-lived compute node. The compute node is used to run the training itself on the V100 GPUs from inside a Docker container. Lastly, Slurm [62] is the job scheduler used on Compute Canada's network.

Docker [63] is used to enable easily reproducible training sessions [64]. It is a lightweight virtualization system, which is used to create distributable and self-contained virtual machines made up of an operating system, software libraries and other dependencies that are isolated from the main operating environment. By using Docker, the code is guaranteed to run using the same drivers, libraries and OS for every training. It also makes the training environment portable since the custom Docker image created for this project could be deployed on other HPC (High-Performance Computing) systems like Amazon's AWS HPC [65]. For this project, the docker image runs Ubuntu 16.04, CUDA driver 10.1 and Python 3.6. All other libraries are defined in the main project's git repository.

Lastly, all the training sessions are logged using Comet.ML [66]. Comet.ML is an online machine learning dashboard that is used to monitor, log and compare training sessions. It can also perform hyperparameter optimization.

Figure 4.5 gives an overview of the full process used to deploy a training session to Compute Canada.

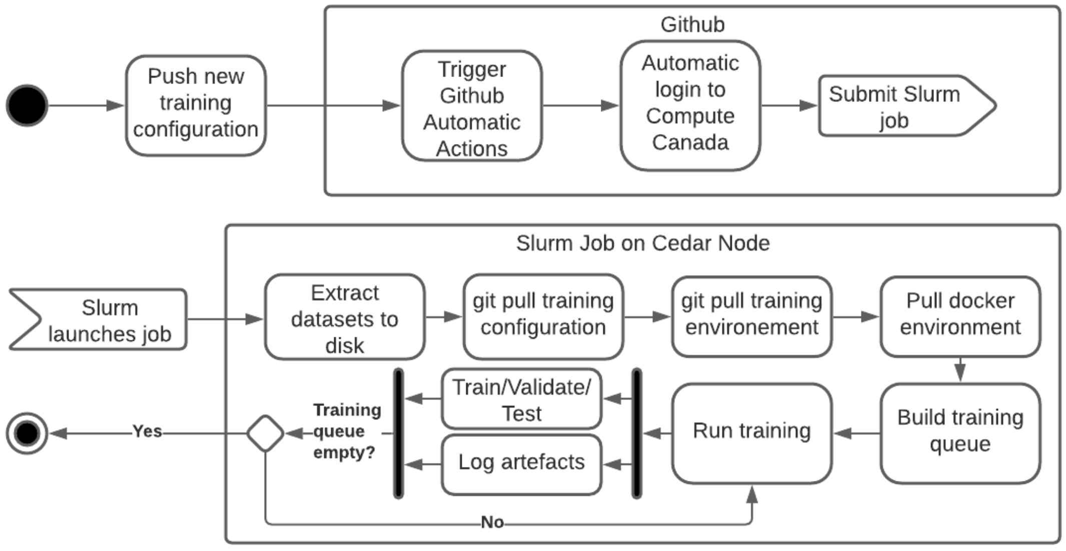


Figure 4.5 UML Activity Diagram [59] of the Automated Training Pipeline

An automated training session is triggered by pushing a new commit to the Github repository containing the training session's configuration files. When pushed, Github will automatically trigger an "Action", which will run an automated script that logs into the Cedar cluster and will submit a new Slurm job request.

Once the requested resources are available, Slurm will run the job's script. The script performs four main actions. First, it will extract the datasets from the persistent storage to the node's disk. Second, it will pull the updated configuration files from the configuration repository. Third, it will pull the project's main git repository to the disk. Lastly, it will pull and iteratively run, in the Docker instance, the different project versions listed in the list of commits.

At this point in the execution procedure, the main project will check whether a hyperparameter search was requested. If it was, it will query Comet.ML for a first set of hyperparameter values, and then it will launch the training session. If it was not, it will simply launch the training session using the default hyperparameter values. Given a compute node can have up to four GPUs, the project can launch up to four parallel training sessions simultaneously. Each one will run isolated from the others, in its own process and on its own GPU.

During the training session, the Comet.ML logger will continuously collect data about the progress of the training session, and log the current training, validation, and test losses.

At the end of all simultaneous training sessions, the program will check whether there are still hyperparameters to test, or whether there are still project commits in the list. If there are, it will launch a new set of training sessions. Otherwise, it will terminate.

4.3.3 Software Testing and Quality Assurance

The training environment is a fairly large project, and training a model involves many moving parts. In order to mitigate the incidence of bugs in the code and thus prevent issues with the training of the models and the data collected, a large set of unit tests were written. These tests guarantee that the software executes as intended. The tests currently cover the following aspects of the training environment:

- Dataset preprocessing
- Dataset segmentation

- Dataset access and data management
- Mathematical correctness of the losses
- Mathematical correctness of geometric transformations
- Mathematical correctness of metrics
- Trajectory reconstruction

The tests are updated and run each time the code is modified to ensure the quality and integrity of the project.

4.4 Training Procedure

This section describes the complete procedure used by this project to train a PyTorch model on the KITTI and Mid-Air datasets.

The training procedure can be broken down into four (4) steps. First, there is the data preparation step. Next is the data acquisition procedure, the training step, and lastly the evaluation step.

4.4.1 Data Preparation

This step prepares the data and the training environment. Although it is desired for the final system to compute the odometry of a vehicle on complete trajectories, it is impractical to train the network on the raw trajectories themselves due to the memory limitation of the system and the issues associated with representing long sequences of data with deep neural networks (Chapter 2, Section 2.2). If we recall the problem statement in Chapter 4, Section 4.1, we see that we only need to learn a function that computes the transformation between two poses of a trajectory. To achieve this, each trajectory only needs to be segmented into short sequences of a couple of frames. A minimum of two frames is required, but more frames allow the network to extract context, which increases prediction accuracy. A hyperparameter search showed that segments of 5 to 7 frames returned the best results. Figure 4.6 shows how a trajectory is segmented.

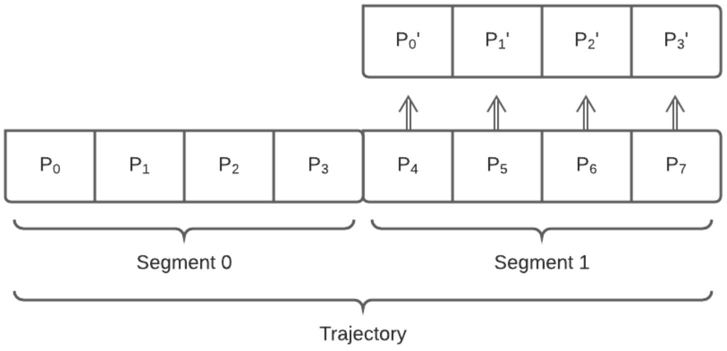


Figure 4.6 Example of the segmentation of a trajectory into segments of 4 frames

Since the ground truth data of each trajectory segment does not start at the origin, the world frame and camera frame must be realigned to the origin from the start of the segment. To do this, all the transformation matrices in the segment must be multiplied by the initial inverse transformation matrix. Moreover, the segment's first frame location vector must be subtracted from all the segment's location vectors. Equation 4.5 describes this procedure:

$$\boldsymbol{P_i}' = \boldsymbol{P_0}^{-1} \begin{bmatrix} \boldsymbol{R}_i & \boldsymbol{t}_i - \boldsymbol{t_0} \\ \boldsymbol{0} & \boldsymbol{1} \end{bmatrix}, \tag{4.5}$$

where $\boldsymbol{P_0} \in \mathbb{R}^{4\times4}$ is the <u>untransformed</u> transformation matrix of the first pose in the segment. $\boldsymbol{R}_i \in \boldsymbol{SO(3)}$ is the $\boldsymbol{i^{th}}$ <u>untransformed</u> rotation matrix in the segment. $\boldsymbol{t_i} \in \mathbb{R}^{3\times1}$ is the $\boldsymbol{i^{th}}$ <u>untransformed</u> translation vector in the segment. Finally, $\boldsymbol{P_i}' \in \mathbb{R}^{4\times4}$ is the $\boldsymbol{i^{th}}$ <u>transformed</u> transformation matrix in the segment. After applying this equation at every time step, we obtain a segment that starts at the origin.

During this step, we also compute the image means of the dataset and standard deviation. These are used to normalize the image's pixel values. The images are also resized to 608×184, in order to reduce their memory footprint.

For the Mid-Air dataset, the ground truth pose is also down sampled from 100 Hz to 25 Hz to match the sampling frequency of the camera. Furthermore, because the Mid-Air dataset frame of reference (NED) is different from the one used by the KITTI dataset, the ground truth data in the body frame will be rotated into the KITTI dataset's camera frame for consistency.

4.4.2 Training

Actual model training is handled by the *ModelTrainer* class. During the training session, data is acquired and fed to the model in minibatches. Each minibatch is built, on the fly, by the PyTorch *Dataloader* and contains a randomly sampled collection of segments. Batching the data allows the training process to take advantage of the high parallelization capabilities of GPUs. It is crucial to properly select the batch size to maximizes memory usage and accuracy. A hyperparameter search found that a minibatch of size $\boldsymbol{N = 15}$ gave optimal results. Although larger minibatches can result in faster training and a more accurate gradient estimate, small batches have a regularizing effect on the model parameters [67], which leads to better generalizations on unseen data [4].

The *ModelTrainer* aims to minimize the inference error, given by a loss function (Chapter 4, Section 4.10), on a minibatch during the training process. It does so through backpropagation and gradient descent. In this project, the *ModelTrainer* uses Adagrad [68] to compute the parameter optimization step. This step updates the weights and biases of the model (Equation 2.1) after the gradient backpropagation is done [4][69]. Adagrad is a common optimizer used in visual odometry learning tasks [40][42] and has led to the most predictable results compared with other popular optimizers tried in this project (Adam [70], RMSProp [71]). Adagrad automatically updates the learning rate for each parameter according to the inverse of the square root of the summed squared historical values of the gradient [4]. A hyperparameter search determined that the optimal starting learning rate for Adagrad was 0.0005.

4.4.3 Validation

At the end of each epoch (a complete pass over the training data), *ModelTrainer* evaluates the state of the model's training by attempting to predict the poses for the segments in the validation split of the datasets. The performance of the model during the evaluation step gives a rough idea of its ability to generalize to unseen data at the current stage of the training process. The same loss function used in the training is used as the evaluation metric for model validation. Model validation data is used to inform early stopping decisions described in Chapter 4, Section 4.6.2.

4.5 Testing Procedure

At the end of the training procedure, the *ModelTester* takes over and evaluates the model's overall performances on the test split of the datasets. Unlike the validation steps of the training procedure, the final model performances are evaluated on the complete test trajectories instead of trajectory segments. *ModelTester* uses an overlapping sliding window mechanism that scans a trajectory and feed as input a sequence of images of predefined size. The resulting predictions are then assembled into a continuous set of poses, using Equation 4.3. Only the $\boldsymbol{n}$ new transformation matrices from the predictions are used.

$$\boldsymbol{n = sliding\ window\ size - the\ number\ of\ overlapping\ frames} \tag{4.6}$$

Picking the right size for the sliding window and the number of overlapping frames is a balancing act between the size of the memory, the inference accuracy of the model and the desired speed of trajectory inference. Large sliding windows require larger memory needs but allow for more data to be used in a single inference pass. This is useful if the accuracy of the model on short sequence is low, because, theoretically, a neural network that can model long-term dependencies can adjust its prediction by observing more of the input sequence. A large frame overlap allows the model to take into consideration more of the past context given new input frames, but it increases the inference time because more frames need to be recomputed. For this project, tests showed that a sliding window size of 30 and an overlap of 15 yielded a good accuracy vs. speed balance. Figure 4.7 shows the sliding window mechanism in action.

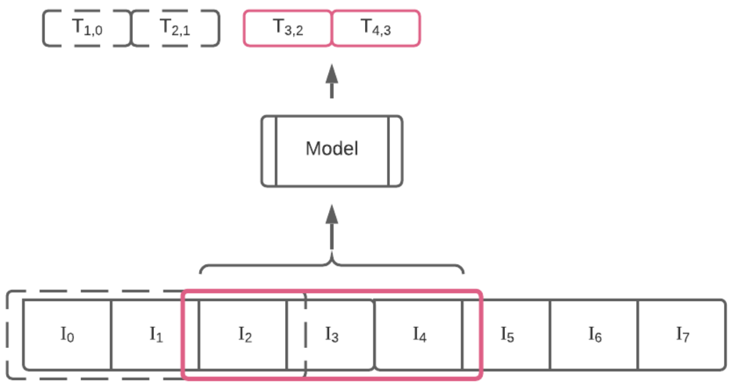


Figure 4.7 Sliding window mechanism — $\boldsymbol{I_i}$ represents an input image at time step $\boldsymbol{i}$ and $\boldsymbol{T_{i+1,i}}$ represent the transformation matrix describing the motion of the camera between time step $\boldsymbol{i}$ and $\boldsymbol{i+1}$.

Once the trajectory poses are computed, they are evaluated using a set of performance metrics commonly found in the visual odometry literature. Performance metrics are computed on both individual trajectories and on the complete set of test trajectories. All metrics are logged to Comet.ML.

4.5.1 KITTI Error Metrics

The KITTI error metrics [53] were introduced along with the KITTI dataset in order to provide standardized tools to compare the performance of different odometry estimation techniques. Those are the metrics used by the KITTI Odometry Leaderboard [20]. The KITTI error metrics are designed to give information about the amount of drift the model will accumulate over a set of discrete distances (100 m, 200 m, etc.) $\boldsymbol{d}$ for both translation and rotation estimates. They are defined as follows:

$$\boldsymbol{Erot(\mathcal{P}_d)} = \frac{\mathbf{1}}{\boldsymbol{d}} \sum_{(\boldsymbol{i},\boldsymbol{j})\in\mathcal{P}_{\boldsymbol{d}}} \angle[(\boldsymbol{\hat{P}_i^{-1}\hat{P}_j})^{-\mathbf{1}}(\boldsymbol{P_i^{-1}P_j})], \tag{4.7}$$

$$\boldsymbol{Etrans(\mathcal{P}_d)} = \frac{\mathbf{1}}{\boldsymbol{d}} \sum_{(\boldsymbol{i},\boldsymbol{j})\in\mathcal{P}_{\boldsymbol{d}}} \left\|(\boldsymbol{\hat{P}_i^{-1}\hat{P}_j})^{-\mathbf{1}}(\boldsymbol{P_i^{-1}P_j})\right\|_{\mathbf{2}} \tag{4.8}$$

where $\boldsymbol{\mathcal{P}_d}$ is a set of pose pairs $\{(\boldsymbol{P_i}, \boldsymbol{P_j}), (\boldsymbol{\hat{P}_i}, \boldsymbol{\hat{P}_j})\}$, $\boldsymbol{i}$ being the starting frame of a segment of distance $\boldsymbol{d}$ and $\boldsymbol{j}$ being the end frame of that segment. $\boldsymbol{P} \in \mathbb{R}^{4\times4}$ is the transformation matrix of the ground truth pose, and $\boldsymbol{\hat{P}} \in \mathbb{R}^{4\times4}$ is the transformation matrix of the estimated pose. $\angle[\,]$ is the angle given by the arccosine of the trace of the rotation matrix contained in the transformation matrix (Equation 4.2). $\|\;\|_2$ is the L2 norm of the translation vector contained in the transformation matrix (Equation 4.2).

The KITTI Error Metrics are evaluated at every starting frame $\boldsymbol{P_{0\to n-1}}$ (Equation 4.4). This can be thought of as a sliding window covering a distance $\boldsymbol{d}$ scanning the whole trajectory. Only the average error per distance for each trajectory is reported.

The KITTI error metrics are considered an improved version of the Relative Error [72]. The project's code uses an adapted version of the python implementation of this metric found in [73].

4.5.2 Absolute Trajectory Error

The Absolute Trajectory Error (ATE) [72] is a popular error metric used to compare two aligned trajectories using the root mean square error (RMSE). It gives an idea of the average deviation of the aligned trajectory from the objective [74] and how well the predicted trajectory shape fits the ground truth. It is defined as follows:

$$\boldsymbol{ATE_{rot}(\mathcal{P})} = \sqrt{\frac{\boldsymbol{1}}{\boldsymbol{n}}\sum_{\boldsymbol{i=0}}^{\boldsymbol{n-1}} \left\|\angle(\boldsymbol{\hat{R}_i R_i}^{-1})\right\|^2} \tag{4.9}$$

$$\boldsymbol{ATE_{pos}(\mathcal{P})} = \sqrt{\frac{\boldsymbol{1}}{\boldsymbol{n}}\sum_{\boldsymbol{i=0}}^{\boldsymbol{n-1}} \|\boldsymbol{t_i} - \boldsymbol{\hat{t}_i}\|^2} \tag{4.10}$$

where $\boldsymbol{\mathcal{P}}$ is the set prediction ($\boldsymbol{\hat{P}_i} \in \mathbb{R}^{4\times4}$) and ground truth ($\boldsymbol{P_i} \in \mathbb{R}^{4\times4}$) pose pairs making the complete trajectory. $\boldsymbol{\hat{R}_i} \in \boldsymbol{SO(3)}$ and $\boldsymbol{R_i} \in \boldsymbol{SO(3)}$ are the aligned estimated and ground truth rotation matrices, respectively. $\boldsymbol{\hat{t}_i} \in \mathbb{R}^{3\times1}$ and $\boldsymbol{t_i} \in \mathbb{R}^{3\times1}$ are the aligned estimated and ground truth translation vectors, respectively. $\angle()$ represent the axis-angle representation of the rotation matrix.

4.6 Regularization

Regularization describes a set of techniques used to help deep neural networks better generalize on unseen data. [4] describes regularization as “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.” For this project, three main regularization techniques are used. Artificial dataset augmentation, early stopping and dropouts.

4.6.1 Data Augmentation

Data augmentation [75] is a regularization technique that involves introducing random variations in the training data in order to reduce the similarity between inputs. This technique mitigates training data overfitting because the variations are significant enough to simulate a dataset that is larger and more varied than it actually is. In this project, random variations in image brightness, saturation and contrast are introduced. Moreover, holes of random sizes are cut out of the image at random locations. Both types of perturbations are randomly applied to the images. The project used an adapted version of the code published with [76].

4.6.2 Early Stopping

Large models with high modelling capacity can have a tendency to overfit the training data. We can observe overfitting in the training and validation loss curves. When overfitting occurs, the training loss will continue to get smaller, but the validation loss will start to increasing instead of decreasing. When we identify this pattern in the validation loss for a number of epochs, it is safe to pre-emptively stop the training session and use the model’s parameters that resulted in the lowest observed validation loss. This project uses an adapted version of the code found in [77]. The code backs up every model parameter resulting in improvements in validation loss and stops the training process after 15 epochs that see no improvement in the loss.

4.6.3 Dropouts

Dropout is a regularization technique that can be seen as introducing noise into the input signal to a neural layer [2]. This is effectively done by randomly severing neural (or “dropping out”) connections between neural layers. To drop a connection, random elements in the input matrix of a layer are set to zero (Equation 2.1) through multiplication with a binary mask (matrix of binary values identifying the elements to drop out). Dropouts prevent the network from strengthening neural connections too quickly and instead force it to explore alternative connections configuration during the training process.

4.7 Hyperparameter Optimization

Hyperparameters are any type of non-learnable parameter that affects a model's behaviour. Learning rate, size of the inference sliding window, size of mini-batch, etc. are all examples of hyperparameters used in this project. Selecting the proper hyperparameters is crucial for the optimal performance of the model. Unfortunately, there is no magic bullet or rule of thumb to select the proper values. Often researchers must rely on previous work, trial and error, and their intuition. However, thanks to the large computing capacity provided by Compute Canada, it was possible to perform searches for many hyperparameters used in this project. Hyperparameter searches are computationally expensive because they require the complete re-training of the model for each new hyperparameter set. However, there exist hyperparameter tuning algorithms that can be used to ease this task. For this project, Comet.ML's Optimizer functionality [78] was used to search for the model's optimal hyperparameters. By giving a selected set of hyperparameters and associated values (a discrete set or a range), the Optimizer can return value combinations selected according to a Bayesian search algorithm [79]. This means that hyperparameters are returned according to their probability of producing a favourable test loss. Appendix A highlights the hyperparameters used in this project.

4.8 Real-Time Inference Utility

The visual odometry of a drone is particularly useful when computed in real time. Odometry information can be used in visual servoing systems [80][81] and to inform decisions in tasks like autonomous drone guidance [82]. In order to facilitate real-time odometry estimation and to satisfy S.O. 2, a real-time inference utility was designed. The utility provides an interface through which a real-time video feed is supplied and a real-time stream of odometry estimation is outputted. The utility was designed to be flexible and is thus based on the FFMPEG library [83]. Using FFMPEG allows for the support of virtually all popular types of inputs. Data can come from raw UDP, TCP/IP, Apple HLS, RMTP, RTSP, etc. The full list of supported video streaming protocols can be found on FFMPEG's website. Inference is performed using the PyTorch model learned during the training process and uses the same sliding window mechanism described in the testing procedure (Chapter 4, Section 4.5) to scan the input feed buffer. The size of the sliding window and the amount of overlap can be configured to ensure real-time performances on a variety of CPUs or GPUs. Currently, odometry estimations are written to a text file and plotted in real-time using matplotlib [84]. However, new output format and destinations can be added by simply sub-classing the *AbstractOutput* class. Figure 4.8 models the activity flow of the utility. Note here that the use of the term "real-time" refers to the ability of the system to run at a frequency high enough to not drop any input frames. The "real-time" performances of this utility are multifactorial and will vary depending on the current load of the system, the speed of the computer, the input stream's framerate, etc. "Real-time" in this context does not refer to the concept of "real-time computing", which guarantees a system's response within specified time constraints.

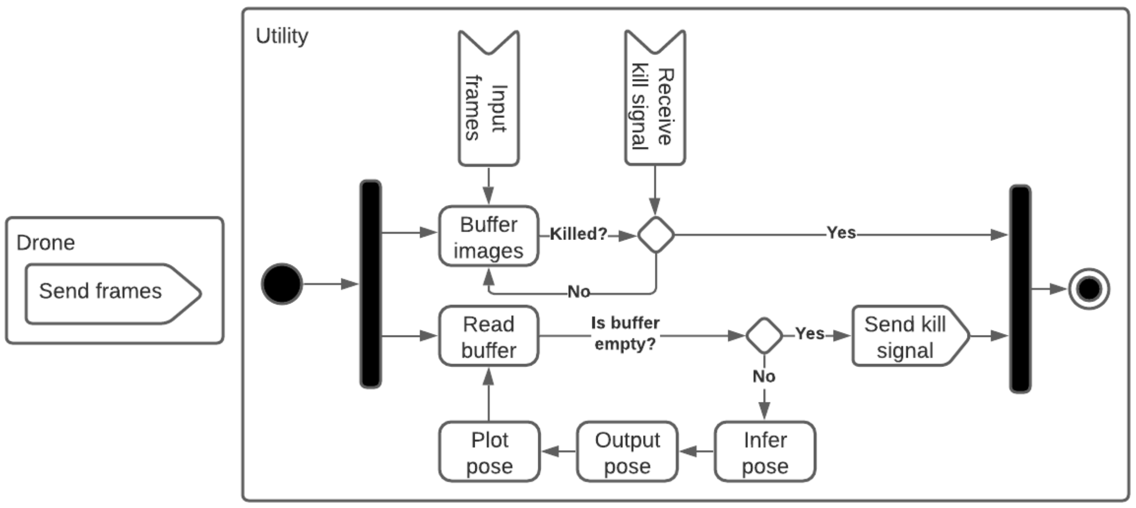


Figure 4.8 UML Activity Diagram of the Inference Utility

4.9 Neural Architecture Design Considerations

Although a single architecture design is presented in Chapter 5, several neural network architectures were considered, trained and tested in the course of this project. The complete list of architecture can be found in Appendix B. All trialled architectures were designed through a semi-systemic process in order to evaluate the validity of hypotheses 1, and 2, cited in the introduction. All architectures implement variations of attention-based network designs found in the literature. Attention-based architectures are becoming increasingly popular for signal processing tasks due to their efficiency. Yet they are under-researched in the field of visual odometry. To the best knowledge of the author, only [44] currently implements an attention-based architecture for visual odometry tasks.

The following design categories were used:

- Architectures combining Bi-RNNs and attention modules
- Architectures replacing RNNs with attention modules
- Architectures, combining Bi-RNNs and attention modules, with an auxiliary task output for global pose estimation

The interest in replacing RNNs layers with attention modules is inspired by the works of [15], [44] and [85]. Using Bi-RNNs to generate context-dependent vector representations of pose transformation was inspired by [17] and [42]. To the best knowledge of the author, coupling contextual pose vector representations generated by Bi-RNNs with an attention module is original to this project. The architectures with auxiliary task output for global pose estimation were inspired by [8] and [43].

4.10 Objective Function Design Considerations

The objective function, also called loss, cost or error function when minimized [4], is a mathematical expression used to describe how far from the desired output the predictions are. It usually returns a single scalar value that is used during gradient descent optimization to identify the model's optimal parameters resulting in the smallest loss (or distance from the desired output) possible.

Throughout this project, a variety of loss functions were implemented and tested in order to identify the best model-loss combination. The complete list of trialled losses can be found in Appendix C. The following design categories were used:

- MSE-based loss
- Temporal geometric consistency-based loss
- Loop closure-based loss
- Compound loss

Mean squared error (MSE) loss is a common type of loss used in machine learning that instructs the gradient descent optimizer to minimize the mean squared error (L2 norm squared) between a predicted and desired output [86].

Temporal geometric consistency losses reward coherence between local pose transformation estimates and global pose estimates using the law of composition for rigid-body transformations. Custom implementations of this type of loss were inspired by the work of Lin et al. [43].

Loop closure-based losses reward the network for identifying loops in the trajectories and correcting the pose estimations accordingly. Although training segments are usually too small to contain loops, the idea is to explicitly teach the network to recognize that similar points of view of a scene should share similar poses. Inspired by the work of Zhang et al. [87], loop closure loss uses a similarity matrix to identify loops (pose similarity) in the ground truth of the segment. The loops are used to further penalize estimates that deviate from the ground truth where a loop should happen.

Compound losses combine two or more losses from the other three categories. Each loss contribution in the compound loss is weighted using a hyperparameter search.

4.11 Model-Loss search

To identify the optimal model and loss combination from the designs discussed in sections 4.9 and 4.10, a search was performed where possible model-loss combinations were trialled. The model-loss search can be thought of as a preselection process that evaluates the performances of a model-loss combination on a subtask to eliminate poorly performing designs. In this case, the model-loss combinations were trained on the KITTI dataset set only. The KITTI dataset was selected over Mid-Air due to its smaller size which allowed for faster training sessions. A model-loss pair was considered a candidate for further tuning, training (on other datasets) and evaluation if it had reached an average validation loss (MSE) below 0.00148 on the KITTI dataset. This is the average optimal validation loss obtained during the model-loss search. Each combination was independently retrained up to five (5) times to account for the non-deterministic reality of stochastic optimization [88]. Due to the large volume of data generated during this search, the large number of model-loss trialled and the fact that most designs resulted in inconclusive results (e.g., no convergence, unstable convergence, too much variability between each training attempts), only the model-loss combination that consistently achieved stable training sessions and a validation MSE below 0.00148 is presented in Chapter 5. A table highlighting the average validation MSE of the different model-loss combination tested can be found in Appendix D.

CHAPTER 5 RESULTS AND DISCUSSION

This chapter presents the best architecture and loss combination identified during the model-loss search described in Chapter 4 Section 4.11.

In order to facilitate the analysis of the results, a PyTorch implementation of DeepVO [40] is used as a benchmark model against, which the results obtained by the candidate model are compared. The DeepVO architecture is trained and tested in the same environment as the candidate model. This eliminates from the data any possible environment-related variations that affect the results reported in the paper (e.g., unreported hyperparameters, framework variations, etc.). DeepVO was chosen because it is a popular model frequently used for performance comparison [8][9][42][43][44] and a PyTorch implementation was readily available online [89].

5.1 Candidate Model and Loss

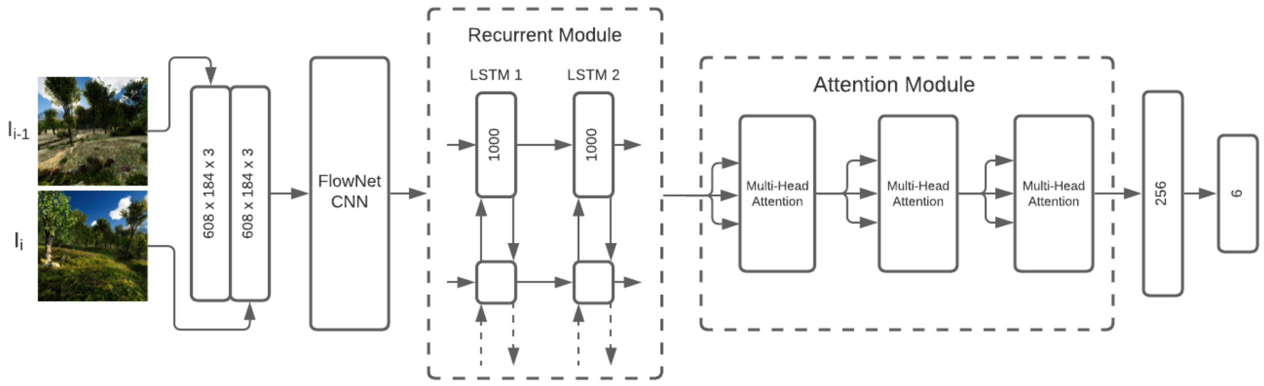


Figure 5.1 SelfAttentionVO architecture overview

The best network architecture identified during the model-loss search described in Chapter 4 Section 4.11 is called SelfAttentionVO. Figure 5.1 shows an overview of the architecture. It was trained using the Mean Squared Error loss on augmented data from the KITTI and Mid-Air datasets. The optimal solution was found after only 21 epochs with early stopping (36 epochs in total). Interestingly, it is one of the simplest designs tried. However, it proved to be more stable and outperformed all the other designs considered, including architecture configurations that used auxiliary tasks, which usually improve the accuracy of estimations of unsupervised networks (Chapter 3, Section 3.2.2).

SelfAttentionVO is composed of four (4) main modules. The convolutional neural network, which extracts features relevant to visual odometry from two (2) stacked consecutive images. A bidirectional LSTM module, which transforms the extracted features into a time-dependent vector representation of the camera's egomotion. An attention module, which adjusts this vector by weighting its component according to the context of the sequences. Lastly, two (2) fully connected linear layers that reduce the multi-dimensional vector into simple 6×1 vectors composed of the camera's rotation and translation.

5.1.1 FlowNet CNN

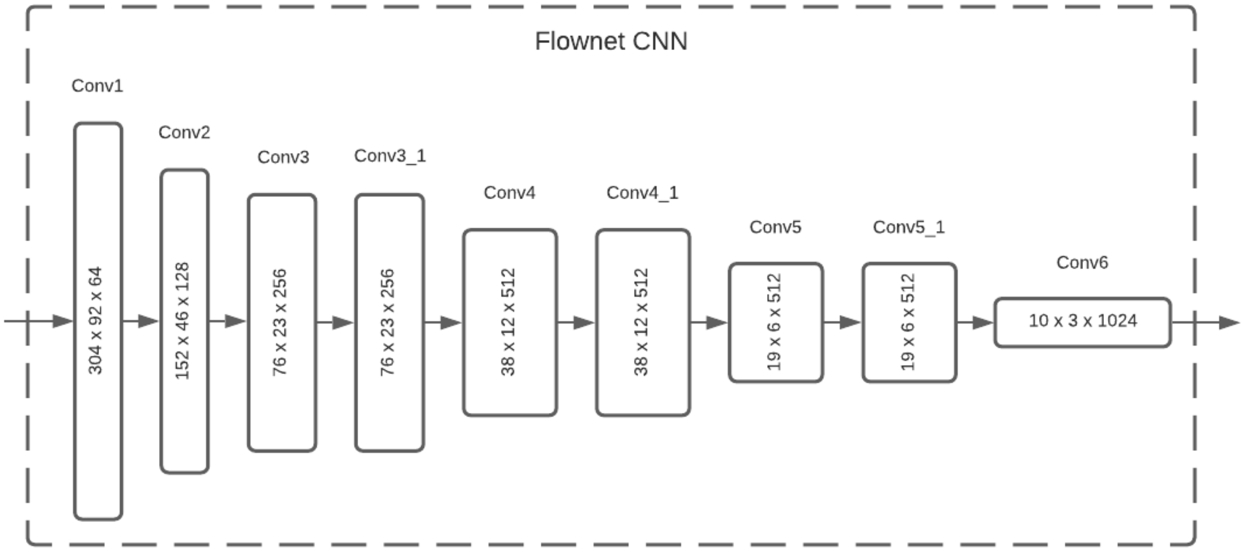


Figure 5.2 FlowNet's convolutional layers

Like [42], [44], [90], [91] and [49] SelfAttentionVO uses the architecture and pre-trained weights of the convolutional encoder of FlowNetSimple [41], shown in Figure 5.2. In FlowNet, these layers extract optical flow features from a stacked pair of images. The idea is that optical flow estimation and ego-motion estimation are related problems and that pixel motion can be used to estimate how a camera moves from one frame to the next [92]. SelfAttentionVO's PyTorch implementation of FlowNet and associated weights comes from [93]. Table 5.1 gives the implementation details of FlowNet's convolutional encoder. Each convolution is followed by a batch normalization [94], *LeakyReLU* and a dropout layer. *LeakyReLU* is an activation function (Equation 2.1) that will set the negative units in the input matrix to a value close to zero. Unlike *ReLU,* this will allow for small gradients to pass through. Table 5.1 describes the implementation details of the convolutional layers of FlowNetSimple.

Table 5.1 FlowNetSimple convolutional layers details

Layer	Kernel Size	Stride	Padding	Input Channels	Output Channels
Conv1	7x7	3	2	6	64
Conv2	5x5	2	2	64	128
Conv3	5x5	2	2	128	256
Conv3_1	3x3	1	1	256	256
Conv4	3x3	1	2	256	512
Conv4_1	3x3	1	1	512	512
Conv5	3x3	1	2	512	512
Conv5_1	3x3	1	1	512	512
Conv6	3x3	1	2	512	1024

5.1.2 Recurrent Module

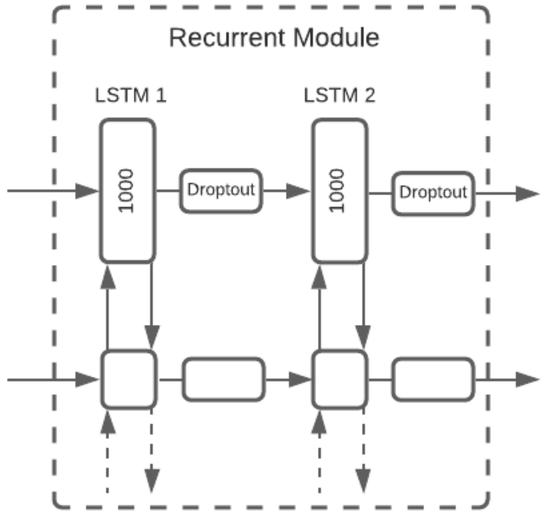


Figure 5.3 Recurrent module

Inspired by [42] and [17], SelfAttentionVO uses two stacked bidirectional LSTM networks as shown in Figure 5.3. These LSTMs transform the extracted CNN features into a high-level vector representation of the camera's motion between time steps. Each vector is context-dependent and is strongly influenced by the hidden state of adjacent LSTM cells. Context-dependent vector representations are crucial to proper odometry estimation. Attempts to replace these layers with a simple fully connected linear layer and relying solely on the attention module for sequence modelling proved unsuccessful. Each layer is composed of a recurrent bidirectional LSTM cell with a hidden state size of 1000 for each direction resulting in output vectors of size 2000. Each layer output is subject to dropouts to aide in network regularization.

5.1.3 Attention Module

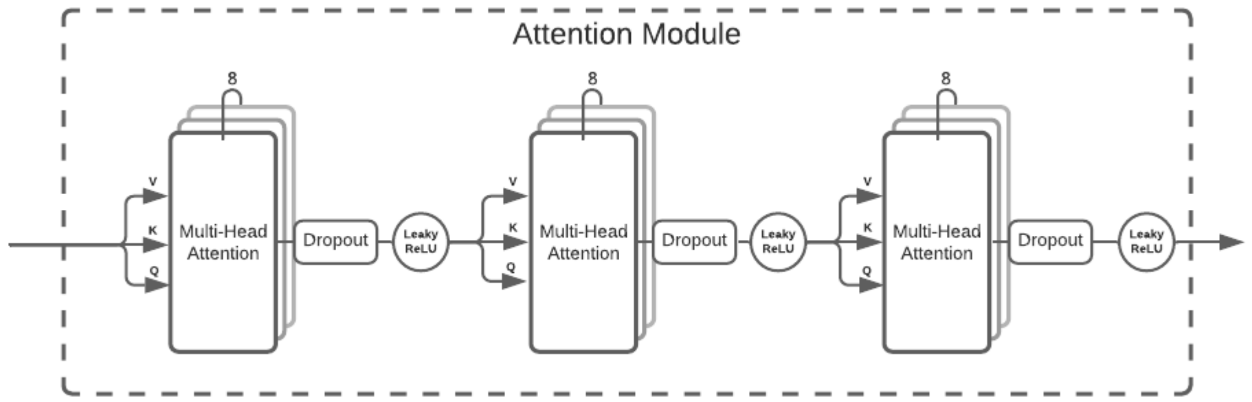


Figure 5.4 Attention module

The attention module, pictured in Figure 5.4, allows SelfAttentionVO to further refine the sequential dependencies modelled by the recurrent module. The module takes as input for $\boldsymbol{V}, \boldsymbol{K}$ and $\boldsymbol{Q}$ the complete sequence of vectors outputted by the bi-LSTMs. As discussed in Chapter 2, Section 2.3, each multi-head attention layer will build a similarity matrix between the vectors. For SelfAttentionVO, the optimal configuration was found to be three (3) multi-head attention layers composed of eight (8) attention heads each. The attention heads will attend information at different positions on the vector. Each multi-head attention layer re-enforces previously modelled sequential decencies. This creates pressure for the network to identify and track patterns in the sequence and tune out the noise in the signal. Dropouts are both used inside the attention heads and after in order to regularize the signal. *LeakyReLU* is used as the activation function after a layer. Other multi-head attention designs (Transformer [15], SNAIL [85]) were considered but proved unsuccessful.

5.1.4 Fully Connected Layers

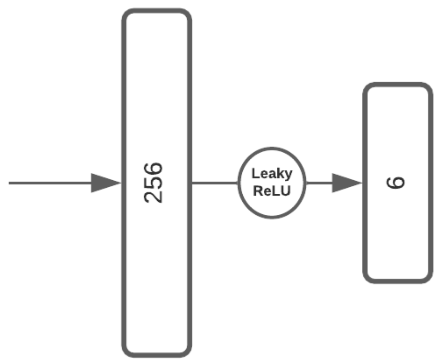


Figure 5.5 Fully connected layers

The fully connected layers, shown in Figure 5.5, are used by SelfAttentionVO to transforms the high-dimensional pose transformation vector, tuned by the attention module, into a 6×1 vector describing the motion between frame $\boldsymbol{I_{i-1}}$ and $\boldsymbol{I_i}$. The first three (3) elements of the output vector represent the rotation in the form of $\boldsymbol{y, x', z''}$ intrinsic Tait-Bryan rotations and the last three (3) elements represent the $\boldsymbol{x, y, z}$ translation. Tait-Bryan angles (radians) are used to represent rotation at the network's output because [40] found that this is preferable to using quaternions, which impedes the training process. However, all angle outputs are converted to quaternions or rotation matrices when performing any mathematical operations.

The use of an intermediary layer that outputs a vector of 256 units allows for a more flexible compression of the attention module's output. This layer is followed by a Dropout mask for regularization and a *LeakyReLU* activation function that introduce non-linearity in the signal.

Other layer configurations were considered, such as an added auxiliary global output for global pose estimation, but they proved unsuccessful.

5.1.5 Candidate Loss

SelfAttentionVO is trained using a Mean Squared Error loss. The Mean Squared Error is arguably one of the most common loss functions used in deep learning. It is equivalent to evaluating the squared Euclidean distance between the estimated vector and the target vector. For the purpose of this project, the MSE between the estimation and the ground truth is separately evaluated on the rotation and translation components of the output vector. The error is estimated over the whole minibatch. The mean error of the rotation and translation estimates are summed, and the rotation estimate is scaled by 100 to increase its weight in the training signal, as is described in [40]. Equation 5.1 describes the procedure:

$$\boldsymbol{Loss_{MSE}} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{j=0}}^{\boldsymbol{N-1}} \left(\sum_{\boldsymbol{i=0}}^{\boldsymbol{n-1}} \mathbf{100} * \|\hat{\boldsymbol{\varphi}}_{\boldsymbol{i}} - \boldsymbol{\varphi}_{\boldsymbol{i}}\|_2^2 + \|\hat{\boldsymbol{t}}_{\boldsymbol{i}} - \boldsymbol{t}_{\boldsymbol{i}}\|_2^2 \right)_{\boldsymbol{j}} \tag{5.1}$$

where $\hat{\boldsymbol{\varphi}}_{\boldsymbol{i}}$ and $\boldsymbol{\varphi}_{\boldsymbol{i}}$ are the predicted and ground truth Tait-Bryan angles, respectively. Other attempts at designing losses that use loop closure constraints or temporal geometric consistency constraints proved unsuccessful.

5.2 Training Results

Convergence toward a solution during training was successfully achieved with SelfAttentionVO on all datasets and data augmentation combinations (KITTI, Mid-Air, Augmented KITTI, Augmented Mid-Air, KITTI & Mid-Air, Augmented KITTI & Mid-Air). A successful convergence is defined as a validation MSE loss that is lower than the corresponding validation loss achieved by the benchmarking model on the same dataset combination.

Figure 5.6 shows that the candidate model, SelfAttentionVO, achieved better (lower) validation losses on the augmented KITTI and Mid-Air datasets compared with DeepVO.

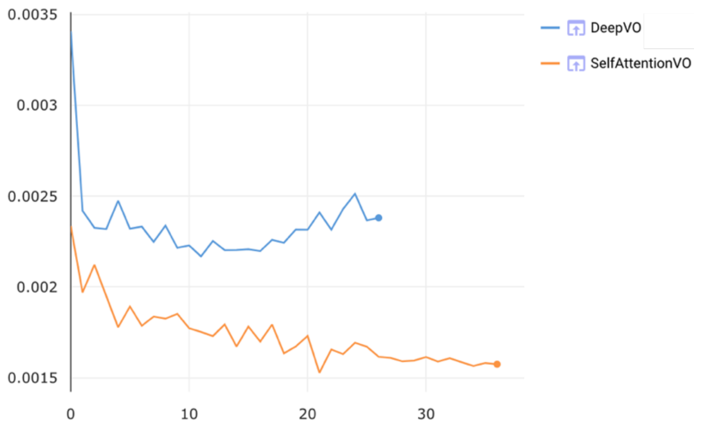


Figure 5.6 Validation MSE loss per epoch for the candidate and benchmark model — The candidate model and the benchmark model are trained on augmented data from the KITTI and Mid-Air datasets.

An analysis of the trends of the training process for the models trained on the KITTI dataset shows that, on average, the performances on the validation dataset are better for SelfAttentionVO compared with DeepVO. Figure 5.7 shows this trend quite clearly. The average validation MSE loss for multiple training sessions of SelfAttentionVO is lower than DeepVO.

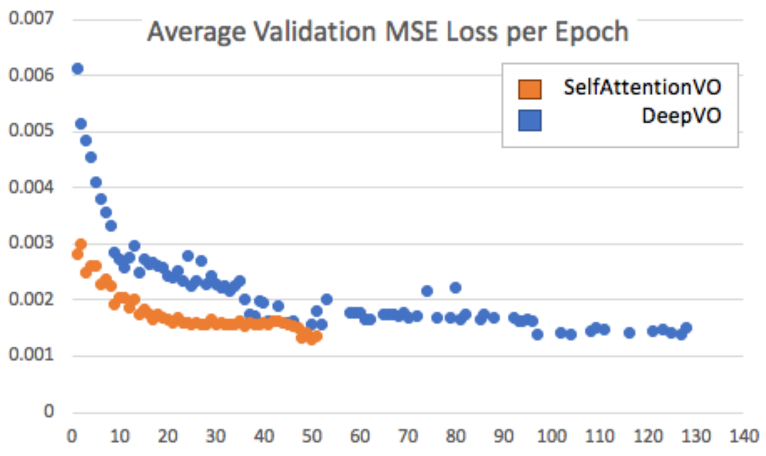


Figure 5.7 Average validation MSE loss per epoch (KITTI dataset) — The mean values are computed from the data of six (6) training sessions for SelfAttentionVO and five (5) training sessions for DeepVO on the KITTI dataset. The best (lowest) average MSE loss for SelfAttentionVO is 0.00143 and 0.00172 for DeepVO.

The trend analysis also found that, on average, solutions on the KITTI dataset were found after 44 epochs for SelfAttentionVO with a 15 epoch early stopping patience. This is 48% faster than DeepVO, which took on average 84 epochs, with early stopping, to converge toward a solution. The trend analysis was performed using models trained on the KITTI dataset since it is smaller, which allowed more training sessions. These results are consistent with what was observed during the training sessions on other dataset configurations.

Early stopping was crucial to prevent overfitting, and it is unclear why it is not used more often in the deep visual odometry literature. Training with a large number of epochs (250), as reported in the DeepVO paper [40], did not yield optimal solutions. In fact, validation loss data shows clear signs of training data overfitting. Figure 5.8 shows that the validation losses start plateauing and even increasing between 20 and 50 epochs for SelfAttentionVO and DeepVO. Although, after more than 200 epochs, the validation losses tend to decrease to levels similar to the ones reached early in the training process, results from the testing datasets do not show better performances than models trained with early stopping. Because of this, there are no obvious benefits in training for large numbers of epochs and early stopping proved to be a good strategy to prevent data overfitting.

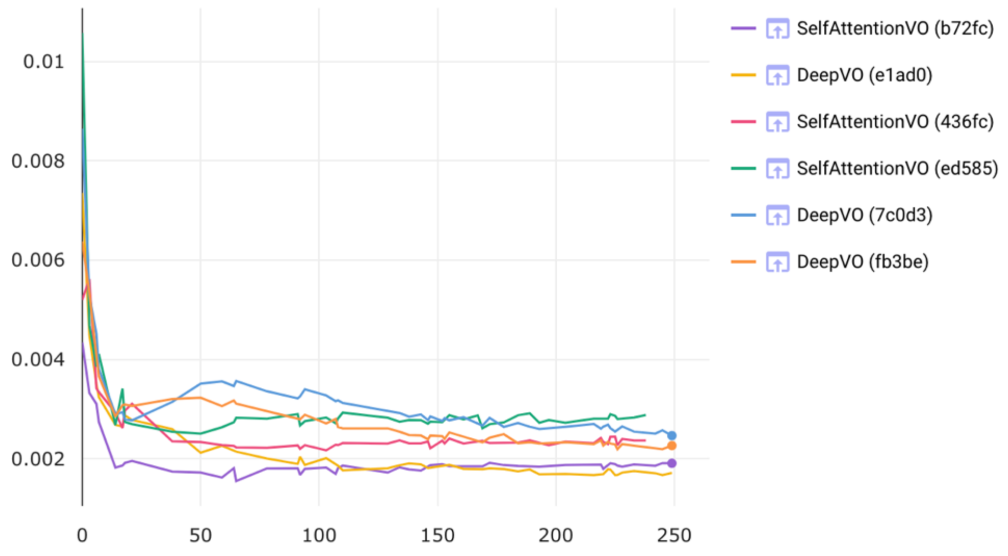


Figure 5.8 Validation MSE loss per epoch for 250 epochs

5.3 Test Results

The optimal solution achieved during training for SelfAttentionVO yields interesting performances on the test sets (non-augmented KITTI and Mid-Air test data). Results show that SelfAttentionVO performs well compared with the benchmark model DeepVO when trained and evaluated under the same conditions. However, both trained DeepVO and SelfAttentionVO underperform during testing when compared with DeepVO's published results. It is unclear what the source of this discrepancy is. Differences in hyperparameter selection may be to blame since DeepVO, like most other papers, does not publish their complete hyperparameter configuration.

The data reported in this section are calculated for test trajectories that have been truncated at 1,000 metres. Beyond 1,000 metres, the accumulated error becomes too great and skews the results. Only Mid-Air trajectories are affected by this measure since all of the test trajectories of this dataset have lengths between 4,973 metres and 6,232 metres.

SelfAttentionVO's performances can be observed in the KITTI translation and rotation error metrics, shown in Figure 5.9 and detailed in Table 5.2. It is clear that the translational and rotational drift of SelfAttentionVO is lower compared with DeepVO with the biggest difference happening when the error is evaluated over 100 metres. This indicates that the local precision of SelfAttentionVO is higher than DeepVO. Note how the impact of an estimation error is more significant on shorter path lengths. This is because a deviation of a few metres over a hundred-metre distance is proportionally more significant than over a few hundred metres. Since the amount of drift is relatively large for both models, error accumulation is significant. However, the accumulation of errors does not grow faster than the length of the trajectory, which is why a downward trend can be observed in the drift percentages on both the translational and rotational error graphs.

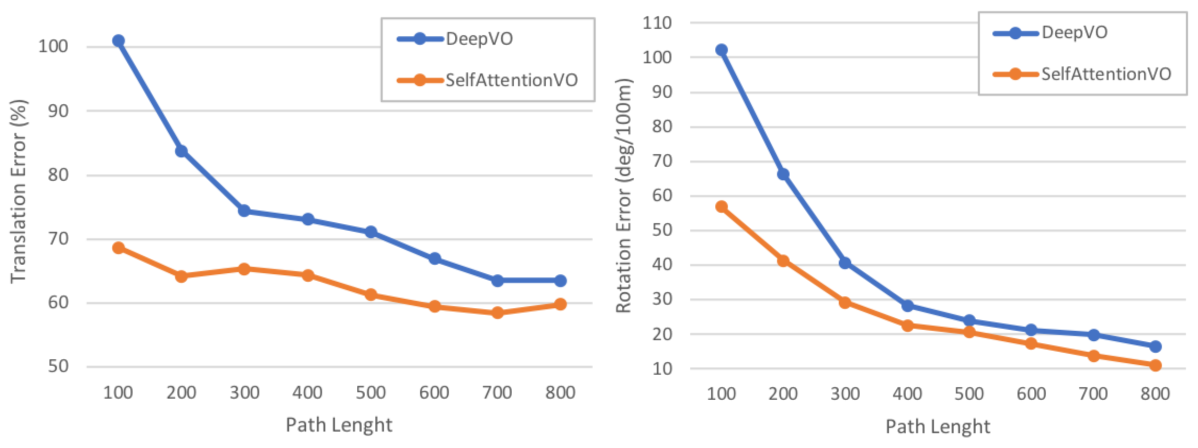


Figure 5.9 Translation and rotation error per path length for DeepVO and SelfAttentionVO

Table 5.2 Detailed average translation and rotation error per path length for DeepVO and SelfAttentionVO

	SelfAttentionVO		DeepVO	
Path Length (m)	Translation Error (%)	Rotation Error (deg/100 m)	Translation Error (%)	Rotation Error (deg/100 m)
100	**68.6**	**56.9**	101	102.1
200	**64.1**	**41.2**	83.8	66.3
300	**65.4**	**29.3**	74.4	40.7
400	**64.4**	**22.6**	73.1	28.4
500	**61.3**	**20.6**	71.1	23.9
600	**59.4**	**17.3**	66.9	21.3
700	**58.4**	**13.8**	63.5	19.9
800	**59.8**	**11.0**	63.5	16.5
Mean	**62.7**	**26.6**	74.7	39.9

Table 5.3 shows that drift is more significant on Mid-Air trajectories compared with KITTI. The errors are especially significant on Mid-Air 1000, 1001 and 1002, which are trajectories in a scene that has a very dense fog. However, SelfAttentionVO outperforms the benchmark model on Translation Error in most cases and always outperforms DeepVO on Rotation Error.

Table 5.3 Detailed average translation and rotation error per path trajectory for DeepVO and SelfAttentionVO

	SelfAttentionVO		DeepVO	
Trajectory	Translation Error (%)	Rotation Error (deg/100 m)	Translation Error (%)	Rotation Error (deg/100 m)
KITTI 07	**20.0**	**9.9**	58.0	49.9
KITTI 10	**23.1**	**9.1**	81.9	47.6
Mid-Air 0000	61.4	**25.4**	**52.6**	45.1
Mid-Air 0001	60.9	**38.7**	**56.5**	57.9
Mid-Air 0002	**52.8**	**26.7**	65.3	38.7
Mid-Air 1000	121.6	**52.7**	**87.5**	56.2
Mid-Air 1001	95.0	**59.0**	149.9	59.5
Mid-Air 1002	85.9	**29.2**	**84.2**	47.1
Mid-Air 2000	**44.2**	**24.8**	64.4	59.2
Mid-Air 2001	**50.2**	**40.3**	66.9	57.4
Mid-Air 2002	**46.2**	**24.6**	80.3	48.4
Mean	**60.1**	**31.0**	77.1	51.5

Although the translational and rotational error differences between SelfAttentionVO and DeepVO are not insignificant, these results do not properly convey how different the reconstructed trajectories actually look like in practice. This is because the KITTI error metrics evaluate the average deviation from the ground truth on specific segment lengths. They do not evaluate how well a predicted trajectory actually fits the ground truth. When doing a qualitative characterization of the trajectories, it can be clearly observed that SelfAttentionVO generates trajectory estimations that have a better fit is compared with DeepVO's benchmark trajectory estimations. The results are most eloquent on the KITTI test trajectories (07, 10) where SelfAttentionVO clearly outperforms DeepVO as is shown in Figure 5.10 (ground truth is in blue; the estimation is in red).

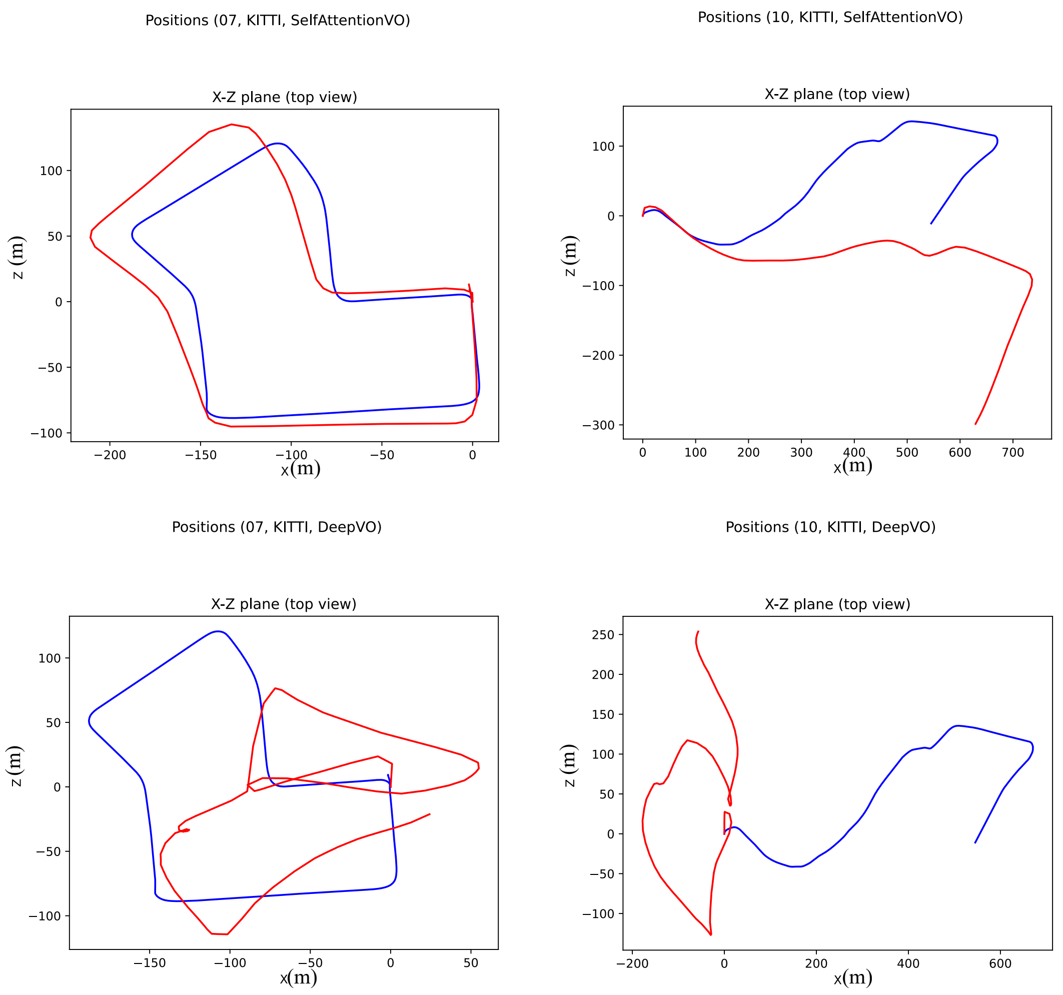


Figure 5.10 Trajectory estimations by SelfAttentionVO and DeepVO for KITTI trajectories 07 and 10 — The trajectories are viewed from the top, with the ground truth in blue and the estimated trajectory in red. The first row of images shows the candidate model SelfAttentionVO and the second row shows DeepVO trained and tested on the same data. Units for the x and z-axis are in metres.

Although less striking, SelfAttentionVO also performs better than the benchmark model on the Mid-Air dataset, as can be seen in Figure 5.11.

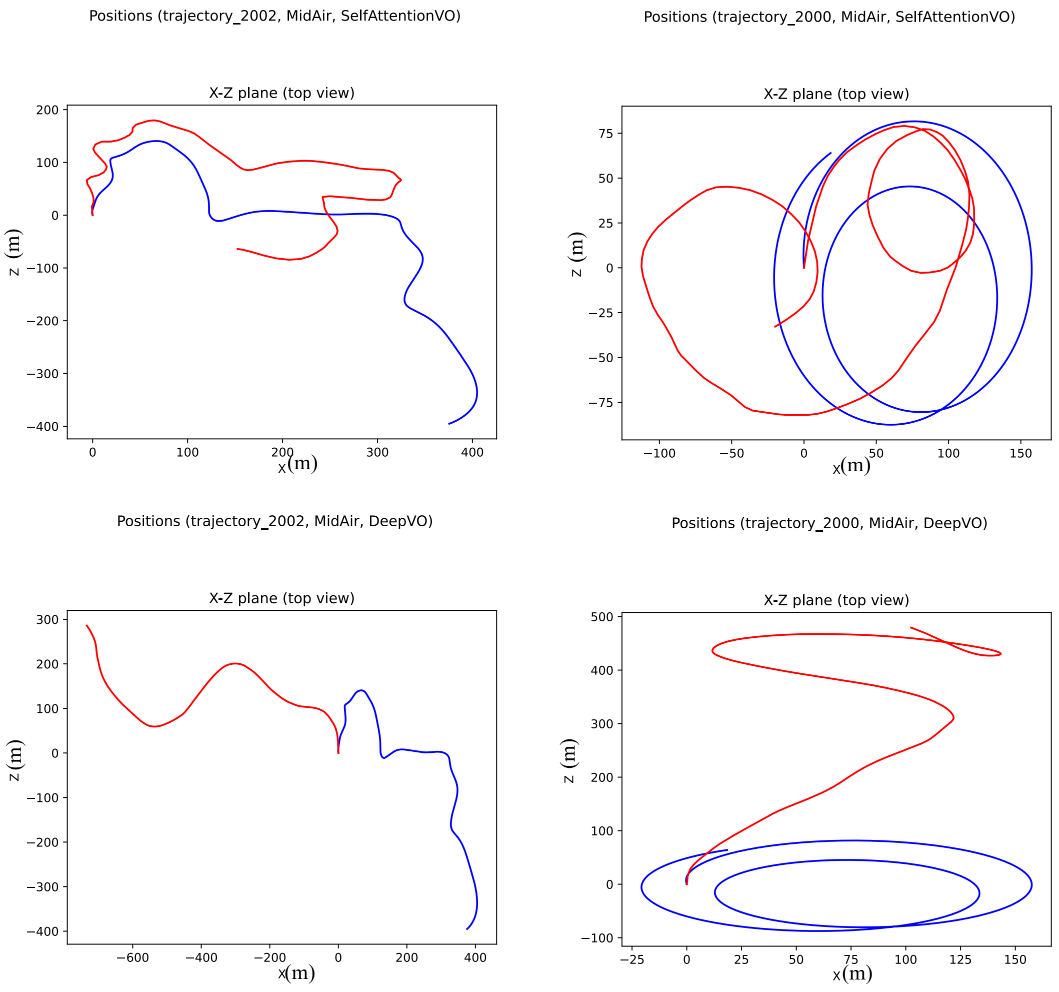


Figure 5.11 Trajectory estimations by SelfAttentionVO and DeepVO for Mid-Air trajectories 2000 and 2002 — The trajectories are viewed from the top, with the ground truth in blue and the estimated trajectory in red. The first row of images shows the candidate model SelfAttentionVO and the second row shows DeepVO trained and tested on the same data. Units for x and z-axis are in metres.

It is important to note here that trajectory reconstruction is most effective in the horizontal (x-z) plane and that any reconstruction in the vertical axis (y) is affected by significant estimation errors for both SelfAttentionVO and DeepVO. It is unclear why this is the case, but it is possible that the lack of vertical motion in the KITTI dataset coupled with the low parallax of the large outdoor nature scenes of Mid-Air has something to do with it. Further research is required to validate or invalidate this hypothesis. Figure 5.12, shows a vertical view of KITTI 07. It can be clearly observed that, for both SelfAttentionVO and the benchmark model, the estimations suffer from significant error accumulation.

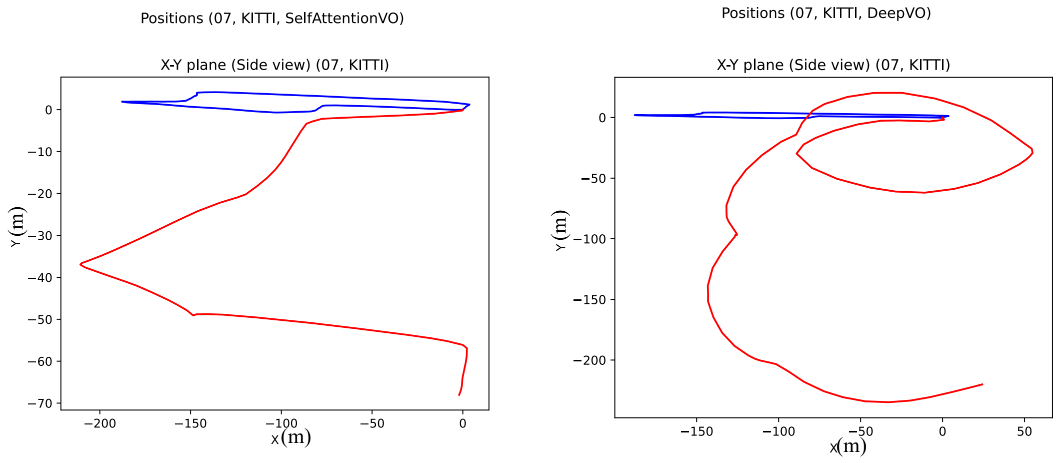


Figure 5.12 Trajectory estimations by SelfAttentionVO for KITTI trajectory 07 (vertical view) — The trajectories are viewed from the side, with the ground truth in blue and the estimated trajectory in red. Units for x and y are in metres.

As discussed in Chapter 3, Section 3.1, low parallax, and points of interests located far from the camera are challenging for visual odometry algorithms, since less data is available for ego-motion estimation [19]. This likely also explains the significant discrepancy between the quality of the odometry estimations on the KITTI dataset versus the Mid-Air dataset. As discussed in Chapter 4, Section 4.2, the KITTI dataset offers videos taken from a car driving around a city. In those kinds of scenes, the points of interest are relatively close to the camera, the movement of the car is predictable and there is limited motion in the vertical axis. On the contrary, Mid-Air's videos are captured by a camera mounted on a drone, the motion of the drone is much less predictable and can be significant in all six (6) degrees of freedom. Moreover, the aerial footage means that most points of interests are distant. Overall, Mid-Air is a significantly more challenging dataset compared with KITTI due to its higher complexity.

The complexity of the Mid-Air dataset can be clearly observed when analyzing the Absolute Trajectory Error. As discussed in Chapter 4, Section 4.5.2, ATE is used to compare two aligned trajectories using the RMSE. Compared with the KITTI Error Metrics, ATE gives a better idea of how well the shape of an estimated trajectory will fit the shape of the ground truth. Figure 5.13 and Table 5.4 show the Absolute Trajectory Errors per trajectories. It can be clearly observed that SelfAttentionVO outperforms the benchmark model for most trajectories. There are significant improvements in SelfAttentionVO's ATE compared with the benchmark model on the KITTI dataset. DeepVO only performed better on Mid-Air 0001, 0002, 2001 and 2002. However on 0001 and 2001 the improvement is only marginal. It is on 0002 and 2002 that the differences are significant, although one must consider that the estimations for these trajectories have large numbers of outliers and variability outside of their upper quartiles.

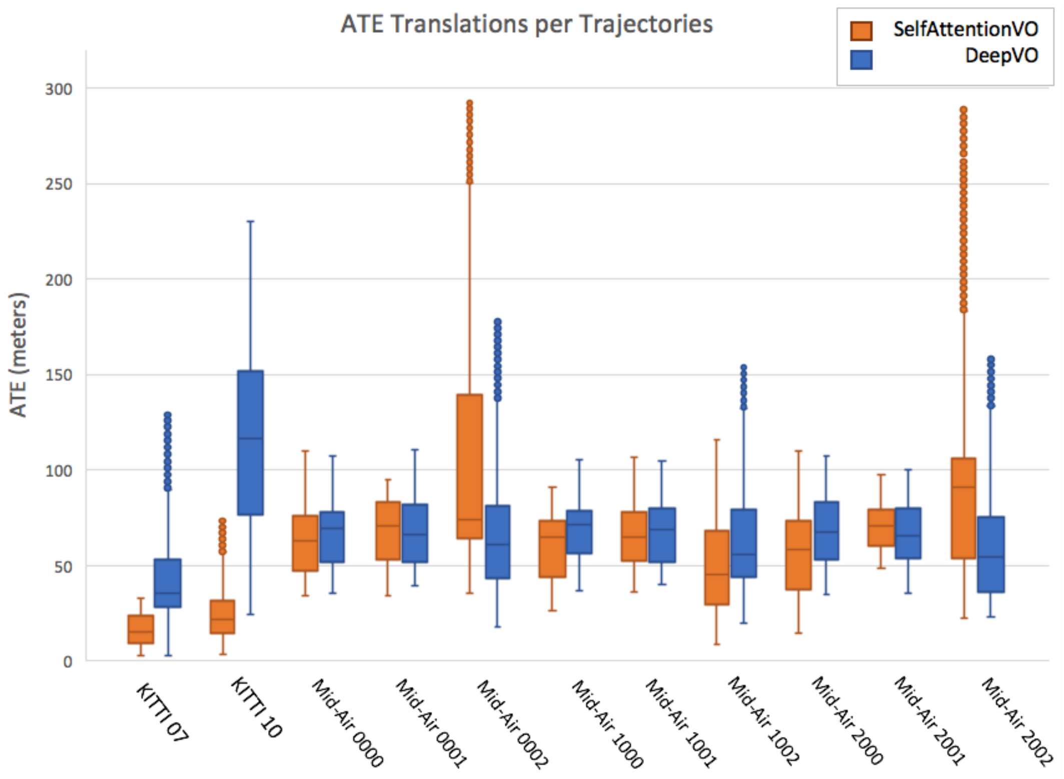


Figure 5.13 Distribution of the Absolute Trajectory Error per test trajectory

Table 5.4 Mean Absolute Trajectory Error per test trajectory

	SelfAttentionVO		DeepVO	
Trajectory	Mean trans. ATE (m)	Mean rot. ATE (deg)	Mean trans. ATE (m)	Mean rot. ATE (deg)
KITTI 07	**16.2**	**16.4**	43.7	87.9
KITTI 10	**27.1**	**21.7**	120.1	118.4
Mid-Air 0000	**62.8**	**87.2**	66.5	113.4
Mid-Air 0001	69.2	136.7	**67.2**	**118.6**
Mid-Air 0002	103.2	**60.4**	**68.2**	91.3
Mid-Air 1000	**60.3**	124.6	68.3	**123.5**
Mid-Air 1001	**65.5**	**126.5**	67.0	131.6
Mid-Air 1002	**48.8**	**86.2**	65.0	113.6
Mid-Air 2000	**56.4**	**58.2**	68.3	125.6
Mid-Air 2001	70.9	**117.7**	**65.5**	130.6
Mid-Air 2002	89.0	**51.0**	**60.0**	111.3
Mean	**60.8**	**80.6**	69.1	115.1

Table 5.5 describes the characteristics associated with each test trajectory. Technically, there are only three (3) different paths in the Mid-Air test set: X000, X0001 and X0002. Mid-Air test trajectories with prefixes 000X, 100X, 200X are environment variations on these three paths. From this table, one can suggest that the reason for the high ATEs on Mid-Air X002 could stem from the higher average velocity at which the vehicle is moving and the larger range covered by the drone in each axis. Unlike X000 and X001 where the drone flies in circles, X002 explores more of the scene (Figure 5.11).

Table 5.5 Test trajectory characteristics

Trajectory	Average Velocity (m/s)	Total Distance (m)	x-axis displacement range	y-axis displacement range	z-axis displacement range	Scene details
KITTI 07	6.3	694.7	[-187.8, 3.7]	[-0.7, 4.2]	[-88.7, 120.6]	Low altitude, City, Day, Sunny
KITTI 10	7.7	919.4	[0, 670.9]	[-15.5, 8.4]	[-41.4, 135.6]	Low altitude, City, Day, Sunny
Mid-Air 0000	11.07	1000	[-20.7, 157.5]	[-20.4, 0.1]	[-87.5, 81.6]	High altitude, Nature, Day, Sunny
Mid-Air 0001	11	1000	[-81.7, 86.5]	[-20.4, 0.1]	[-20.7, 156.6]	High altitude, Nature, Day, Sunny
Mid-Air 0002	13.2	1000	[-0, 404.9]	[-134.2, 0]	[-395.2, 140.6]	High altitude, Nature, Day, Sunny
Mid-Air 1000	11.07	1000	[-20.7, 157.5]	[-20.4, 0.1]	[-87.5, 81.6]	High altitude, Nature, Day, Foggy
Mid-Air 1001	11	1000	[-81.7, 86.5]	[-20.4, 0.1]	[-20.7, 156.6]	High altitude, Nature, Day, Foggy
Mid-Air 1002	13.2	1000	[-0, 404.9]	[-134.2, 0]	[-395.2, 140.6]	High altitude, Nature, Day, Foggy
Mid-Air 2000	11.07	1000	[-20.7, 157.5]	[-20.4, 0.1]	[-87.5, 81.6]	High altitude, Nature, Day, Sunset, Cloudy
Mid-Air 2001	11	1000	[-81.7, 86.5]	[-20.4, 0.1]	[-20.7, 156.6]	High altitude, Nature, Day, Sunset, Cloudy
Mid-Air 2002	13.2	1000	[-0, 404.9]	[-134.2, 0]	[-395.2, 140.6]	High altitude, Nature, Day, Sunset, Cloudy

Interestingly, unlike the KITTI Error Metrics, the fog present in Mid-Air 100X does not seem to negatively impact the ATE errors for SelfAttentionVO. This result is surprising and suggests that, although fog in a scene will cause significant drift, it will not have a significant impact on the overall shape of the trajectory. As shown in Figure 5.14, the fog in those scenes is quite dense. Only highly contrasted points of interests located close to the camera can be perceived. It can be hypothesized that deep visual odometry models have a tendency to extract ego-motion information from features located closer to the camera, but global positioning information from more distant features.

Figure 5.14 Preview of Mid-Air trajectory 1000

It is clear that SelfAttentionVO's performances are better than the benchmark model. Overall, SelfAttentionVO allows for around 22% reduction in mean translational drift (KITTI Translation Error) and 40% reduction in mean rotational drift (Rotation Error) when calculated on complete trajectories (capped at 1,000 metres). Moreover, translational fit is improved by about 12% (translation ATE) and rotational fit is improved by about 30% (rotation ATE).

5.4 Ablation Study

The training of the candidate SelfAttentionVO model used multiple techniques to achieve the results presented in Section 5.3. This section aims to characterize the contribution of these techniques to the improvements in results observed in Section 5.3. To do this, different training protocols were tested to explore the contributions of each technique. All the training protocols used the same test data as Section 5.3, non-augmented KITTI and Mid-Air, except for Table 5.7 where the networks are trained on non-augmented data but tested on augmented data.

Table 5.6 Model performances given different training procedures

Model	Dataset	Augmented Training Data	Avg KITTI Translation Error (%)	Avg KITTI Rotation Error (deg/100 m)	Avg Translation ATE (m)	Avg KITTI Rotation ATE (deg)
SelfAttentionVO	KITTI	False	89.6	37.0	73.4	105.3
DeepVO	KITTI	False	102.4	47.9	73.5	116.1
SelfAttentionVO	Mid-Air	False	59.5	39.7	72.4	97.7
DeepVO	Mid-Air	False	64.4	51.6	74.6	117.7
SelfAttentionVO	KITTI	True	121.5	34.4	74.8	97.5
DeepVO	KITTI	True	146.4	48.7	69.6	110.1
SelfAttentionVO	Mid-Air	True	57.8	39.9	77.2	117.6
DeepVO	Mid-Air	True	69.0	43.9	62.6	100.1
SelfAttentionVO	KITTI, Mid-Air	False	**49.6**	39.7	67.0	97.5
DeepVO	KITTI, Mid-Air	False	92.3	52.6	83.0	130.4
SelfAttentionVO	KITTI, Mid-Air	True	60.1	**31**	**60.8**	**80.6**
DeepVO	KITTI, Mid-Air	True	77.1	51.5	69.1	115.1

Table 5.6 shows the performances of DeepVO and SelfAttentionVO when trained on KITTI, on Mid-Air, on KITTI and Mid-Air and on the augmented version of these datasets.

Training on Mid-Air improves the performances of both networks compared with when they are only trained on KITTI. This is likely due to the bigger size of Mid-Air compared with KITTI since more data means more examples from which the networks can learn.

As expected, training using both KITTI and Mid-Air has, also, a positive impact on the performances of both networks. Again, more data usually means better learning.

Surprisingly, on its own, the effects of data augmentation are mostly negative or marginal for the networks trained on individual datasets. This makes sense since the type of data augmentation performed can be compared with data corruption (punching holes, jittery pixel values). However, given enough data, the networks become more robust to these types of corruption. A general improvement in performance can be observed when the models are trained on augmented data from both the KITTI and Mid-Air datasets.

Table 5.7 Model performance on corrupted data

Model	Dataset	Augmented Test Data	Avg KITTI Translation Error (%)	Avg KITTI Rotation Error (deg/100 m)	Avg Translation ATE (m)	Avg KITTI Rotation ATE (deg)
SelfAttentionVO	KITTI	True	**61.2**	**53.7**	**73.8**	**121.9**
DeepVO	KITTI	True	96.3	55.4	77.9	126.9

Table 5.7 shows the performance of DeepVO and SelfAttentionVO when trained on non-augmented data but evaluated on augmented data. Since the type of data augmentation used is akin to data corruption, this table evaluates the performances of both networks on corrupted data. It can be seen that SelfAttentionVO appears to be significantly more robust to noisy/corrupted data than DeepVO.

Table 5.8 Model performance given deactivated RNN or attention module

Modification	Dataset	Augmented Training Data	Avg KITTI Translation Error (%)	Avg KITTI Rotation Error (deg/100 m)	Avg Translation ATE (m)	Avg KITTI Rotation ATE (deg)
Removed Attention Module	KITTI	False	181.1	43.7	74.2	**103.5**
Replaced RNN with Linear Layer	KITTI	False	**90.3**	**41.0**	**66.7**	112.8

Table 5.8 evaluates the individual contributions of the RNN and Attention module used in SelfAttentionVO. In one case, the attention module was deactivated in order to calculate the contribution of the RNN, and, in the other case, the RNN was replaced by a simple fully connected layer of size 2000 in order to calculate the contribution of the attention module. It is clear that both modules work synergistically to provide the performances reported in Chapter 5, Section 5.3. However, unlike the RNN, the attention module alone can do much of the heavy lifting, improving both drift and fit. This suggests that the attention module is better than the RNN at modelling the sequential dependencies of visual odometry tasks.

Table 5.9 Impacts of the over-representation of the Mid-Air training data on KITTI test data inference

Model	Test Trajectory	Training Dataset	Augmented Data	Avg KITTI Translation Error (%)	Avg KITTI Rotation Error (deg/100 m)	Avg Translation ATE (m)	Avg KITTI Rotation ATE (deg)
SelfAttentionVO	07	KITTI	False	25.0	11.8	**23.8**	**20.6**
SelfAttentionVO	07	KITTI, Mid-Air	False	**22.8**	**10.6**	24.1	26.1
SelfAttentionVO	10	KITTI	False	34.7	10.6	42.9	29.7
SelfAttentionVO	10	KITTI, Mid-Air	False	**25.1**	**7.5**	**16.4**	**28.9**

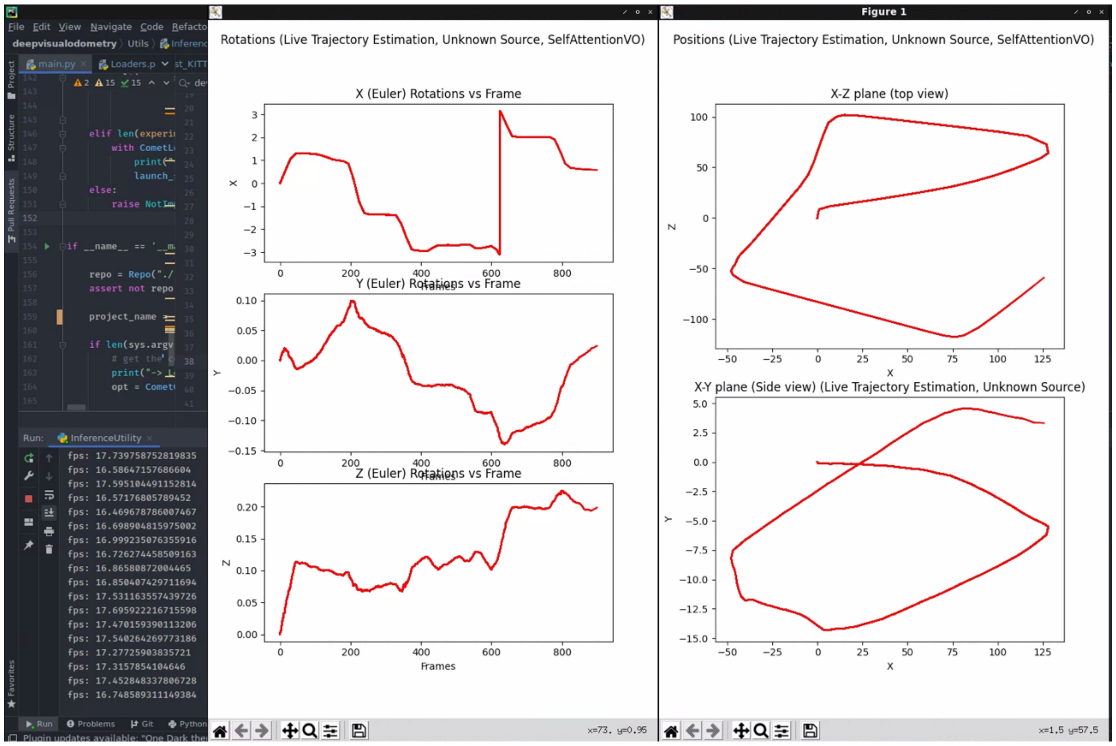
Lastly, an interesting outcome of adding an extra dataset is the lack of a negative impact on the prediction of the KITTI dataset. This result is surprising because the Mid-Air dataset is composed of significantly more images. This means that it is overrepresented in the training data compared with the KITTI dataset. The expected side effect of such imbalance in the training data would have been a reduction in accuracy on the KITTI dataset compared with a model trained exclusively on the underrepresented dataset. Yet, this is not what is observed in Table 5.9. On the contrary, most testing metrics on the test dataset slightly improved. The extra training data appears to be large and varied enough to regularize the training process and counteract any possible imbalance without the need to implement any implicit imbalance mitigation techniques.

5.5 Inference Utility Performances

As mentioned in Chapter 4, Section 4.8, a real-time inference utility was designed so that the candidate model can be used on live video feeds. To be usable, the utility should perform pose inference at a frame rate high enough to not drop any frames from the input feed.

Inference using a deep neural network is usually fast. This is because neural networks can essentially be broken down into a series of matrix multiplications, which are highly parallelizable. However, because of their sequential nature, networks using RNNs have a larger execution time, since recurrence cannot be parallelized.

Nonetheless, it is still possible to use SelfAttentionVO for real-time inference. With a sliding window of size 30 and overlap of 15, tests on a pre-recorded video (KITTI 07) streamed over HLS and UDP showed that, on average, the inference utility can process approximately 15 frames per second on the CPU, which is enough to process videos from the KITTI dataset. When running the utility on the GPU, the frame rate jumps up to 60 frames per second, which is more than enough to process videos from the Mid-Air dataset. The CPU used for these tests is an Intel Core i9-9900K running at 3.60 GHz, and the GPU is an Nvidia RTX 2080 Ti.


File Edit View Navigate Code Refacto
deepvisualodometry 〉 Utils 〉 Inferenc
main.py
Loaders.p
Run: InferenceUtility
Run
Problems
Git
Python
Plugin updates available: "One Dark the
Rotations (Live Trajectory Estimation, Unknown Source, SelfAttentionVO)
X (Euler) Rotations vs Frame
Y (Euler) Rotations vs Frame
Z (Euler) Rotations vs Frame
Frames
x=73. y=0.95
Figure 1
Positions (Live Trajectory Estimation, Unknown Source, SelfAttentionVO)
X-Z plane (top view)
X-Y plane (Side view) (Live Trajectory Estimation, Unknown Source)
x=1.5 y=57.5

```
        elif len(experi
            with CometL
                print("
                launch_
        else:
            raise NotIm

if __name__ == '__m

    repo = Repo("./
    assert not repo

    project_name =

    if len(sys.argv
        # get the' c
        print("-> L
        opt = Comet
```

```
fps: 17.739758752819835
fps: 16.58647157686604
fps: 17.595104491152814
fps: 16.57176805789452
fps: 16.469678786007467
fps: 16.698904815975002
fps: 16.999235076355916
fps: 16.726274458509163
fps: 16.86580872004465
fps: 16.850407429711694
fps: 17.531163557439726
fps: 17.695922216715598
fps: 17.470159390113206
fps: 17.540264269773186
fps: 17.27725903835721
fps: 17.3157854104646
fps: 17.452848337806728
fps: 16.748589311149384
```

Figure 5.15 Screenshot of the Live Inference Utility estimating KITTI 07 streamed over UDP on the CPU

Figure 5.15 shows the inference utility displaying a live plot of the inferred trajectory. The inferred translations are plotted for both the vertical (x-y) and horizontal plane (x-z). For the rotations, the utility displays a plot for each axis. The average frame rate at which the utility runs is printed to the standard output.

Since the sliding window configurations for live inference are the same as the ones used for the model testing, the estimation accuracy using the inference utility is technically the same as the accuracy observed during the testing phase. However, as can be seen in Figure 5.15, the visualized trajectory is different compared with the images of Trajectory 07 presented in Figure 5.10. This is due to the fact the inference is started a few seconds after the stream is started since the stream needs to exist for the utility to work properly.

These tests show that the inference utility is a viable way of performing visual odometry estimation on a live video feed.

5.6 Discussion

The conclusive results presented in this chapter demonstrate that developing an intelligent system capable of estimating the visual odometry of a drone, in real time, is indeed possible. The initial objective and sub-objective defined in Chapter 1, Section 1.2 were successfully reached. Namely, the design of SelfAttentionVO, a deep neural network capable of estimating the pose difference between two input images, satisfies S.O.1. The design of the inference utility, a utility capable of performing visual odometry estimation in real time using SelfAttentionVO, satisfies S.O.2. Lastly, the results presented in Chapter 5, Sections 5.3 and 5.4, satisfy S.O.3.

Out of all stated hypotheses, only one was shown to be partially false. Indeed, it does not appear to be the case that an auxiliary global pose estimation task can increase the accuracy of visual odometry estimates (Hypothesis 2). During the training and hyperparameter research process, no model-loss combination showed better performances than SelfAttentionVO trained using a MSE loss. This hypothesis was marked as partially false because the literature review clearly shows that auxiliary tasks are viable, especially for unsupervised networks. However, in the case of a supervised network, it seems that the training signal created by the MSE loss provides enough information required for the training process. On the contrary, architectures and losses enforcing global and relative geometrically consistent poses appear not to offer significantly more training information than the MSE. Given the ground truth data naturally encodes those geometric constraints, it is hypothesized that an MSE loss on the ground truth data also encodes these constraints in the training signal. Since no other type of auxiliary tasks were attempted, these conclusions are only limited to auxiliary global pose estimations. The results associated with these attempts are not included in Chapter 5 because the data is often incomplete or inconsistent due to their failed training.

The other three hypotheses (Hypotheses 1, 3 and 4) were successfully validated. First, multi-head attention modules do seem to increase the accuracy of visual odometry estimations (Hypothesis 1). The ablation study shows a reduction in drift of about 50% and an improvement in fit of about 10% when the attention module is used to model sequential dependencies instead of the RNN module. Second, the dataset expansion techniques used (adding an extra dataset, image augmentation) proved essential to achieving the current level of accuracy reported for SelfAttentionVO (Hypothesis 3). Lastly, it does seem possible to use the inference utility in real time given proper sliding window and hardware configurations, this supports Hypothesis 4.

One of the most unexpected and positive results of SelfAttentionVO is the significant improvement in convergence speed. As stated in Chapter 5, Section 5.2, compared with DeepVO, it takes on average 48% fewer epochs for the training session to converge to a solution. Even if the performances of SelfAttentionVO had been equivalent to those of DeepVO, the improvement in training time is in itself a huge achievement and would certainly warrant more research in the use of attention modules for visual odometry tasks.

Although the results are very promising, there are still significant shortcomings and issues that need to be addressed. The biggest issue is the significant reduction in performance compared with the results reported in the literature. Even with the unit tests, which guarantee the correctness of the code used in the training environment, such discrepancy is unusual. Given a difference in hyperparameter configuration is suspected, further tuning would be required to validate or invalidate this hypothesis.

A second issue that needs to be addressed is the significant error accumulation in the vertical axis for both SelfAttentionVO and DeepVO. As stated in Section 5.3, this may be caused by the lack of vertical motion or the low parallax of some scenes in the datasets. Further research is required to validate or invalidate this hypothesis.

Another area of uncertainty is the real-world performances of the model. Because the model was trained and tested on simulated data (Mid-Air) and non-aerial data (KITTI), it is probable that the performances will not translate well to real aerial footage. Characterization of real-world performances would first require supplementary tests on a non-simulated aerial dataset, such as the EuRoC MAV dataset [95]. If the performances of the candidate model on a test split of the EuRoC dataset are deemed sufficient, then real-world experiments using the real-time inference utility could be performed.

The current solution does not implement any drift correction techniques. Unfortunately, without drift correction, it is unlikely that a model like SelfAttentionVO could compete with the current state-of-the-art techniques [20]. Although attempts were made to learn loop closure as described in Chapter 4 Section 4.10, the memory requirements proved to be too great. To effectively perform loop closure the model would be required to ingest the complete trajectory in one pass. Future work using SelfAttentionVO should consider coupling the model with an external loop closure algorithm that can work in real time [96], [97].

Although the solution is designed to be used with drones, there is technically no specific architectural limitation that would prevent SelfAttentionVO to work with other types of vehicles provided it is trained and tuned on the appropriate data. Given a properly trained model, the inference utility can be used with any type of video input, be it a drone, car or anything else for which we wish to compute visual odometry.

The inference utility is designed to be a viable alternative to running the model directly on the drone's onboard computer. This decision was made because the level of model optimization required to be able to run on resource-limited hardware was outside the scope of this project. Nonetheless, the performance of the inference utility could still be improved. One of the biggest enhancements would be to remove the LSTM layers from SelfAttentionVO. Although Chapter 5, Section 5.4 clearly shows the need for an LSTM to create context dependent-vector embeddings, it does not change the fact that LSTMs are computationally less efficient due to their sequential nature. Alternative ways of efficiently creating these vectors should be researched. Apart from removing the LSTM layers, other enhancements can be made without touching SelfAttentionVO's architecture. Namely, the model can be pruned and quantized. In a nutshell, pruning removes any unnecessary neural connections in the model and quantization reduces the number of bits representing each connection. Not only do these optimizations improve computational speed, but they also significantly reduce the memory footprint of the model. Wang et al. observed a reduction of the size of models from up to 49 × and an increase in computational efficiency of up to 4 × [98]. Considering that the current storage requirements of the model's weights are 1.3 Gb, these improvements are non-negligible. Not only would the improvements increase the performances of the inference utility, but they would likely be significant enough to allow the model to be deployed on a Jetson Nano [99], a small general-purpose on-board computer that can be used to run deep neural models on small vehicles like drones. Using an optimized model on a Jetson would be the last step before redesigning the model to run as a completely embedded system.

CHAPTER 6 CONCLUSION

The goal of this research project was to design an intelligent system that can estimate the visual odometry of a drone in real time. The solution to this objective was reached by developing a real-time inference utility that uses a deep neural network to estimate the visual odometry from any type of live video streaming protocol.

Along with the real-time inference utility, the main contribution to the field of visual odometry research is the novel architecture of the deep neural network. The neural network combines a convolutional neural network, a recurrent neural network, an attention module and fully connected layers to extract the camera's ego-motion from a sequence of images. The convolutional network extracts visual features relevant to visual odometry, the recurrent network vectorizes those features into time-dependent vectors, the attention module tunes those vectors according to the context of the sequence and, finally, the fully connected layers compress those vectors into ego-motion predictions. The combination of the attention module and recurrent network to model the sequential dependencies is, to the best knowledge of the author, original to this research. The network was trained on the KITTI and the Mid-Air dataset using a mean squared error loss. Test results showed that the architecture significantly reduces the mean translational drift by 20% and improves the mean translational fit of the estimated trajectory by 12% compared with DeepVO, which is an equivalent neural network often used as a benchmark in the literature. Furthermore, the ablation study showed that the attention module is essential to improve the accuracy of visual odometry estimation compared with the benchmark model. It does so by increasing the robustness of the network to noise in the signal. SelfAttentionVO is 36% more robust to translational drift and has a 5% better translational fit than DeepVO when given noisy data. The ablation study also showed that a large dataset combined with data augmentation techniques provide a significant reduction in both KITTI Error Metrics (30% reduction in Translation Error) and Absolute Trajectory Errors (17% reduction in translational ATE). Lastly, training data shows that early stopping is a good and underused strategy in the field of deep visual odometry to prevent the network from overfitting the training data. When training, SelfAttentionVO converges 48% faster than DeepVO.

Certain conditions apply to the results obtained by SelfAttentionVO. Namely, further tuning will be required to bring the performance to the levels of accuracy reported in the literature. Moreover, further testing on non-simulated aerial data is required to characterize the performance of the network on real drone footage.

Strategies to improve the network's accuracy should be centred around loop closing and auxiliary task optimization. Although these strategies have proved unsuccessful in the course of this project, the current literature clearly shows that they could greatly improve the quality of the network's predictions.

While the network can already be used for real-time inference through the inference utility, it cannot be deployed, in its current form, as an embedded system. Future work should focus on optimizing the run time performances of the network so that it can run in resource-limited environments.

REFERENCES

[1] R. I. Popescu, M. Raison, G. M. Popescu, D. Saussié, and S. Achiche, “Design and development of a novel type of table tennis aerial robot player with tilting propellers,” *Mechatronics*, vol. 74, p. 102483, Apr. 2021, doi: 10.1016/j.mechatronics.2021.102483.

[2] C. Coulombe, J.-F. Gamache, O. Barron, G. Descôteaux, D. Saussié, and S. Achiche, “Task Taxonomy for Autonomous Unmanned Aerial Manipulator: A Review,” presented at the ASME 2020 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Nov. 2020. doi: 10.1115/DETC2020-22297.

[3] M. W. Gardner and S. R. Dorling, “Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences,” *Atmos. Environ.*, vol. 32, no. 14–15, pp. 2627–2636, Aug. 1998, doi: 10.1016/S1352-2310(97)00447-0.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European conference on computer vision*, 2014, pp. 818–833.

[7] K. R. Konda and R. Memisevic, “Learning visual odometry with a convolutional network.,” in *VISAPP (1)*, 2015, pp. 486–490.

[8] A. Valada, N. Radwan, and W. Burgard, “Deep Auxiliary Learning for Visual Localization and Odometry,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 6939–6946. doi: 10.1109/ICRA.2018.8462979.

[9] N. Radwan, A. Valada, and W. Burgard, “VLocNet++: Deep Multitask Learning for Semantic Visual Localization and Odometry,” *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 4407–4414, Oct. 2018, doi: 10.1109/LRA.2018.2869640.

[10] Y. Bengio, P. Frasconi, and P. Simard, “The problem of learning long-term dependencies in recurrent networks,” in *IEEE International Conference on Neural Networks*, Mar. 1993, pp. 1183–1188 vol.3. doi: 10.1109/ICNN.1993.298725.

[11] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, 1994.

[12] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] J. Kleenankandy and K. A. A. Nazeer, “An enhanced Tree-LSTM architecture for sentence semantic modeling using typed dependencies,” *Inf. Process. Manag.*, vol. 57, no. 6, p. 102362, Nov. 2020, doi: 10.1016/j.ipm.2020.102362.

[14] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997, doi: 10.1109/78.650093.

[15] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[16] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *ArXiv14090473 Cs Stat*, May 2016, Accessed: Jan. 20, 2021. [Online]. Available: http://arxiv.org/abs/1409.0473

[17] M. E. Peters *et al.*, “Deep contextualized word representations,” *ArXiv180205365 Cs*, Mar. 2018, Accessed: Jan. 20, 2021. [Online]. Available: http://arxiv.org/abs/1802.05365

[18] M. Irani, B. Rousso, and S. Peleg, *Recovery of ego-motion using image stabilization*. Leibniz Center for Research in Computer Science, Department of Computer …, 1993.

[19] D. Scaramuzza and F. Fraundorfer, “Visual odometry [tutorial],” *IEEE Robot. Autom. Mag.*, vol. 18, no. 4, pp. 80–92, 2011.

[20] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun, “Visual Odometry / SLAM Evaluation 2012,” *The KITTI Vision Benchmark Suite*. http://www.cvlibs.net/datasets/kitti/eval_odometry.php (accessed Oct. 10, 2019).

[21] H. P. Moravec, “Obstacle avoidance and navigation in the real world by a seeing robot rover.,” Stanford Univ CA Dept of Computer Science, 1980.

[22] L. Matthies and S. Shafer, “Error modeling in stereo navigation,” *IEEE J. Robot. Autom.*, vol. 3, no. 3, pp. 239–248, 1987.

[23] C. Engels, H. Stewénius, and D. Nistér, “Bundle adjustment rules,” *Photogramm. Comput. Vis.*, vol. 2, no. 2006, 2006.

[24] H. Strasdat, J. Montiel, and A. J. Davison, “Scale drift-aware large scale monocular SLAM,” *Robot. Sci. Syst. VI*, vol. 2, no. 3, p. 7, 2010.

[25] D. Nistér, “Preemptive RANSAC for live structure and motion estimation,” *Mach. Vis. Appl.*, vol. 16, no. 5, pp. 321–329, 2005.

[26] D. G. Lowe, “Object recognition from local scale-invariant features.,” in *iccv*, 1999, vol. 99, pp. 1150–1157.

[27] H. Bay, T. Tuytelaars, and L. Van Gool, “Surf: Speeded up robust features,” in *European conference on computer vision*, 2006, pp. 404–417.

[28] D. Nister, “An efficient solution to the five-point relative pose problem,” in *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, 2003, vol. 2, p. II–195.

[29] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, “Bundle adjustment—a modern synthesis,” in *International workshop on vision algorithms*, 1999, pp. 298–372.

[30] M. Irani and P. Anandan, “About direct methods,” in *International Workshop on Vision Algorithms*, 1999, pp. 267–277.

[31] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, “DTAM: Dense tracking and mapping in real-time,” in *2011 international conference on computer vision*, 2011, pp. 2320–2327.

[32] J. Engel, T. Schöps, and D. Cremers, “LSD-SLAM: Large-scale direct monocular SLAM,” in *European conference on computer vision*, 2014, pp. 834–849.

[33] J. R. Bergen, P. Anandan, K. J. Hanna, and R. Hingorani, “Hierarchical model-based motion estimation,” in *European conference on computer vision*, 1992, pp. 237–252.

[34] B. K. Horn and B. G. Schunck, “Determining optical flow,” *Artif. Intell.*, vol. 17, no. 1–3, pp. 185–203, 1981.

[35] Udacity, “Optical Flow Constraints,” 2015. https://www.youtube.com/watch?v=-XXGrsFE4UI (accessed Feb. 09, 2021).

[36] N. Yang, R. Wang, X. Gao, and D. Cremers, “Challenges in monocular visual odometry: Photometric calibration, motion bias, and rolling shutter effect,” *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 2878–2885, 2018.

[37] I. Cvišic, J. Cesic, I. Markovic, and I. Petrovic, “Soft-slam: Computationally efficient stereo visual slam for autonomous uavs,” *J. Field Robot.*, 2017.

[38] R. Mur-Artal and J. D. Tardos, “ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras,” *IEEE Trans. Robot.*, vol. 33, no. 5, pp. 1255–1262, Oct. 2017, doi: 10.1109/TRO.2017.2705103.

[39] X. Yin, X. Wang, X. Du, and Q. Chen, “Scale recovery for monocular visual odometry using depth estimated with deep convolutional neural fields,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 5870–5878.

[40] S. Wang, R. Clark, H. Wen, and N. Trigoni, “Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 2043–2050.

[41] A. Dosovitskiy *et al.*, “Flownet: Learning optical flow with convolutional networks,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2758–2766.

[42] J. Jiao, J. Jiao, Y. Mo, W. Liu, and Z. Deng, “MagicVO: An End-to-End Hybrid CNN and Bi-LSTM Method for Monocular Visual Odometry,” *IEEE Access*, vol. 7, pp. 94118–94127, 2019, doi: 10.1109/ACCESS.2019.2926350.

[43] Y. Lin *et al.*, “Deep global-relative networks for end-to-end 6-DoF visual localization and odometry,” in *Pacific Rim International Conference on Artificial Intelligence*, 2019, pp. 454–467.

[44] E. Parisotto, D. Singh Chaplot, J. Zhang, and R. Salakhutdinov, “Global Pose Estimation With an Attention-Based Recurrent Network,” 2018, pp. 237–246. Accessed: May 14, 2020. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2018_workshops/w9/html/Parisotto_Global_Pose_Estimation_CVPR_2018_paper.html

[45] T. Zhou, M. Brown, N. Snavely, and D. G. Lowe, “Unsupervised learning of depth and ego-motion from video,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1851–1858.

[46] R. Szeliski, “Prediction Error as a Quality Metric for Motion and Stereo,” Sep. 1999, Accessed: Jan. 21, 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/prediction-error-as-a-quality-metric-for-motion-and-stereo/

[47] R. Li, S. Wang, Z. Long, and D. Gu, “Undeepvo: Monocular visual odometry through unsupervised deep learning,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 7286–7291.

[48] Y. Li, Y. Ushiku, and T. Harada, “Pose Graph Optimization for Unsupervised Monocular Visual Odometry,” in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 5439–5445.

[49] G. Iyer, J. Krishna Murthy, G. Gupta, M. Krishna, and L. Paull, “Geometric Consistency for Self-Supervised End-to-End Visual Odometry,” 2018, pp. 267–275. Accessed: Sep. 02, 2020. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018_workshops/w9/html/Iyer_Geometric_Consistency_for_CVPR_2018_paper.html

[50] N. Yang, L. von Stumberg, R. Wang, and D. Cremers, “D3VO: Deep Depth, Deep Pose and Deep Uncertainty for Monocular Visual Odometry,” 2020, pp. 1281–1292. Accessed: Sep. 02, 2020. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2020/html/Yang_D3VO_Deep_Depth_Deep_Pose_and_Deep_Uncertainty_for_Monocular_CVPR_2020_paper.html

[51] M. Klodt and A. Vedaldi, “Supervising the new with the old: learning sfm from sfm,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 698–713.

[52] P. Domingos, “Every Model Learned by Gradient Descent Is Approximately a Kernel Machine,” *ArXiv Prepr. ArXiv201200152*, 2020.

[53] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? The KITTI vision benchmark suite,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, Providence, RI, Jun. 2012, pp. 3354–3361. doi: 10.1109/CVPR.2012.6248074.

[54] M. Fonder and M. Van Droogenbroeck, “Mid-Air: A Multi-Modal Dataset for Extremely Low Altitude Drone Flights,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Long Beach, CA, USA, Jun. 2019, pp. 553–562. doi: 10.1109/CVPRW.2019.00081.

[55] “RT3000 v3 » GNSS-aided inertial navigation system for automotive testing,” *OxTS*. https://www.oxts.com/products/rt3000/ (accessed Feb. 15, 2021).

[56] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun, “Sensor Setup,” *The KITTI Vision Benchmark Suite*. http://www.cvlibs.net/datasets/kitti/setup.php (accessed Feb. 15, 2021).

[57] “The HDF5® Library & File Format,” *The HDF Group*. https://www.hdfgroup.org/solutions/hdf5/ (accessed Feb. 15, 2021).

[58] “PyTorch documentation — PyTorch 1.6.0 documentation.” https://pytorch.org/docs/1.6.0/ (accessed Feb. 27, 2021).

[59] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[60] “Compute Canada - Calcul Canada,” *Compute Canada - Calcul Canada*. https://www.computecanada.ca (accessed Feb. 16, 2021).

[61] “Cedar - CC Doc.” https://docs.computecanada.ca/wiki/Cedar (accessed Feb. 16, 2021).

[62] A. B. Yoo, M. A. Jette, and M. Grondona, “Slurm: Simple linux utility for resource management,” in *Workshop on job scheduling strategies for parallel processing*, 2003, pp. 44–60.

[63] Docker, “Empowering App Development for Developers.” https://www.docker.com/ (accessed Feb. 17, 2021).

[64] C. Boettiger, “An introduction to Docker for reproducible research,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, 2015.

[65] Amazon Web Services, Inc., “High Performance Computing (HPC),” *AWS*. https://aws.amazon.com/hpc/ (accessed Feb. 17, 2021).

[66] Comet.ML, “Comet.ML home page.” https://www.comet.ml/ (accessed Feb. 17, 2021).

[67] D. R. Wilson and T. R. Martinez, “The general inefficiency of batch training for gradient descent learning,” *Neural Netw.*, vol. 16, no. 10, pp. 1429–1451, 2003.

[68] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization.,” *J. Mach. Learn. Res.*, vol. 12, no. 7, 2011.

[69] PyTorch, “torch.optim — PyTorch 1.7.1 documentation.” https://pytorch.org/docs/stable/optim.html (accessed Feb. 19, 2021).

[70] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *ArXiv14126980 Cs*, Jan. 2017, Accessed: Feb. 19, 2021. [Online]. Available: http://arxiv.org/abs/1412.6980

[71] G. Hinton, N. Srivastava, and K. Swersky, “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,” *Cited On*, vol. 14, no. 8, 2012.

[72] Z. Zhang and D. Scaramuzza, “A tutorial on quantitative trajectory evaluation for visual (-inertial) odometry,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 7244–7251.

[73] H. Zhan, *Huangying-Zhan/kitti-odom-eval*. 2021. Accessed: Feb. 20, 2021. [Online]. Available: https://github.com/Huangying-Zhan/kitti-odom-eval

[74] S. Umeyama, “Least-squares estimation of transformation parameters between two point patterns,” *IEEE Comput. Archit. Lett.*, vol. 13, no. 04, pp. 376–380, 1991.

[75] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” *ArXiv161103530 Cs*, Feb. 2017, Accessed: Feb. 18, 2021. [Online]. Available: http://arxiv.org/abs/1611.03530

[76] S. Zhao, Z. Liu, J. Lin, J.-Y. Zhu, and S. Han, “Differentiable Augmentation for Data-Efficient GAN Training,” *ArXiv200610738 Cs*, Jul. 2020, Accessed: Sep. 02, 2020. [Online]. Available: http://arxiv.org/abs/2006.10738

[77] B. M. Sunde, *Bjarten/early-stopping-pytorch*. 2021. Accessed: Feb. 23, 2021. [Online]. Available: https://github.com/Bjarten/early-stopping-pytorch

[78] Comet.ML, “Optimizer.” https://www.comet.ml/docs/python-sdk/Optimizer/ (accessed Feb. 23, 2021).

[79] Comet.ML, “Bayes Algorithm.” https://www.comet.ml/docs/python-sdk/introduction-optimizer/#bayes-algorithm (accessed Mar. 26, 2021).

[80] A. Mohebbi, S. Achiche, and L. Baron, “Integrated and concurrent detailed design of a mechatronic quadrotor system using a fuzzy-based particle swarm optimization,” *Eng. Appl. Artif. Intell.*, vol. 82, pp. 192–206, Jun. 2019, doi: 10.1016/j.engappai.2019.03.025.

[81] A. Mohebbi, S. Achiche, and L. Baron, “Multi-criteria fuzzy decision support for conceptual evaluation in design of mechatronic systems: a quadrotor design case study,” *Res. Eng. Des.*, vol. 29, no. 3, pp. 329–349, Jul. 2018, doi: 10.1007/s00163-018-0287-6.

[82] Alexandre Duperré, “Guidage et planification réactive de trajectoire d’un drone monoculaire contrôlé par intelligence artificielle,” Polytechnique Montréal, 2020. [Online]. Available: https://publications.polymtl.ca/5399/

[83] “FFmpeg.” http://ffmpeg.org/ (accessed Feb. 25, 2021).

[84] “Matplotlib: Python plotting — Matplotlib 3.3.4 documentation.” https://matplotlib.org/stable/index.html (accessed Feb. 25, 2021).

[85] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel, “A Simple Neural Attentive Meta-Learner,” *ArXiv170703141 Cs Stat*, Feb. 2018, Accessed: Jun. 01, 2020. [Online]. Available: http://arxiv.org/abs/1707.03141

[86] “MSELoss — PyTorch 1.7.1 documentation.” https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html (accessed Feb. 27, 2021).

[87] X. Zhang, L. Wang, Y. Zhao, and Y. Su, “Graph-Based Place Recognition in Image Sequences with CNN Features,” *J. Intell. Robot. Syst.*, vol. 95, no. 2, pp. 389–403, Aug. 2019, doi: 10.1007/s10846-018-0917-2.

[88] J. C. Spall, *Introduction to Stochastic Search and Optimization*, 1st ed. USA: John Wiley & Sons, Inc., 2003.

[89] ChiWeiHsiao, *ChiWeiHsiao/DeepVO-pytorch*. 2021. Accessed: Mar. 05, 2021. [Online]. Available: https://github.com/ChiWeiHsiao/DeepVO-pytorch

[90] V. Mohanty, S. Agrawal, S. Datta, A. Ghosh, V. D. Sharma, and D. Chakravarty, “Deepvo: A deep learning approach for monocular visual odometry,” *ArXiv Prepr. ArXiv161106069*, 2016.

[91] B. Ummenhofer *et al.*, “Demon: Depth and motion network for learning monocular stereo,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5038–5047.

[92] G. Costante, M. Mancini, P. Valigi, and T. A. Ciarfuglia, “Exploring representation learning with cnns for frame-to-frame ego-motion estimation,” *IEEE Robot. Autom. Lett.*, vol. 1, no. 1, pp. 18–25, 2015.

[93] C. Pinard, *ClementPinard/FlowNetPytorch*. 2021. Accessed: Mar. 08, 2021. [Online]. Available: https://github.com/ClementPinard/FlowNetPytorch

[94] “BatchNorm2d — PyTorch 1.6.0 documentation.” https://pytorch.org/docs/1.6.0/generated/torch.nn.BatchNorm2d.html?highlight=batchnorm2d #torch.nn.BatchNorm2d (accessed Mar. 22, 2021).

[95] M. Burri *et al.*, “The EuRoC micro aerial vehicle datasets,” *Int. J. Robot. Res.*, vol. 35, no. 10, pp. 1157–1163, 2016.

[96] A. Angeli, S. Doncieux, J. Meyer, and D. Filliat, “Real-time visual loop-closure detection,” in *2008 IEEE International Conference on Robotics and Automation*, May 2008, pp. 1842–1847. doi: 10.1109/ROBOT.2008.4543475.

[97] D. Bai, C. Wang, B. Zhang, X. Yi, and X. Yang, “CNN Feature boosted SeqSLAM for Real-Time Loop Closure Detection,” *ArXiv170405016 Cs*, Apr. 2017, Accessed: Mar. 19, 2021. [Online]. Available: http://arxiv.org/abs/1704.05016

[98] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *ArXiv Prepr. ArXiv151000149*, 2015.

[99] “New Jetson Nano 2GB Developer Kit,” *NVIDIA*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/education-projects/ (accessed Mar. 19, 2021).

.

APPENDIX A HYPERPARAMETERS

Table A.1 Hyperparameters used in this project

Name	Value	Note
Segment Length	5-7 frames	Found using a hyperparameter search
Minibatch Size	15 segments	
Adagrad’s Learning Rate	0.0005	
Early Stopping Patience	15 epochs	
Sliding Window Size	30 frames	
Sliding Window Overlap	15 frames	
Input Image Width	608 pixels	Used the same values as [89]
Input Image Height	184 pixels	
KITTI Images Mean (R,G,B)	(0.35034978101920455, 0.36983621966485647, 0.36422201692803685)	Computed values
KITTI Images Standard Deviation (R,G,B)	(0.3153969452509568, 0.31948839594698725, 0.3234648542300365)	
Mid-Air Images Mean (R,G,B)	(0.5040787380191862, 0.515162692666903, 0.45762643440828815)	
Mid-Air Images Standard Deviation (R,G,B)	(0.2813783349205511, 0.2697398916118524, 0.30578316760563706)	

APPENDIX B TESTED MODEL ARCHITECTURES

Table B.1 SelfAttentionVO architecture details

SelfAttentionVO
Note: Candidate model
INPUT: Concatenated image pair sequence
FlowNetSimple Convolutional Encoder
LSTM (hidden_size = 1000, num_layers=2, bidirectional=True)
Dropout (p=0.5, inplace=False)
Multi-Head Attention 1 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Multi-Head Attention 2 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Multi-Head Attention 3 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=2000, out_features=256, bias=True)
LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=256, out_features=6, bias=True)
OUTPUT: Relative pose transformation vector

Table B.2 SimpleSelfAttentionVO architecture details

SimpleSelfAttentionVO
Note: SelfAttentionVO with the LSTM layer replaced by a linear layer
INPUT: Concatenated image pair sequence
FlowNetSimple Convolutional Encoder
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=30720, out_features=2000, bias=True)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Multi-Head Attention 1 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Multi-Head Attention 2 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Multi-Head Attention 3 (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
Dropout (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=2000, out_features=256, bias=True)
LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=256, out_features=6, bias=True)
OUTPUT: Relative pose transformation vector

Table B.3 GlobalRelativeSelfAttentionVO architecture details

GlobalRelativeSelfAttentionVO	
Note: This experimental design loosely inspired by [15] attempted to use an auxiliary global pose output to create an intermediate multidimensional global pose vector that could be used as a Query in the common multi-head attention module. The idea was to let the Common Multi-Head Attention module learn to model the relation between global pose estimates and relative pose transformation estimates. The hope was that it would use this relationship to tune relative pose transformation estimates according to how well they assembled into global pose estimates. This design does not perform well and was quickly discarded.	
INPUT: Concatenated image pair sequence	
FlowNetSimple Convolutional Encoder	
LSTM (hidden_size = 1000, num_layers=2, bidirectional=True)	
Dropout (p=0.5, inplace=False)	
Global Multi-Head Attention (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)	**Relative Multi-Head Attention 1** (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
INTERMEDIATE OUTPUT: Intermediate global pose multidimensional vector	**-**
Dropout (p=0.5, inplace=False)	**Dropout** (p=0.5, inplace=False)
LeakyReLU (negative_slope=0.1, inplace=True)	**LeakyReLU** (negative_slope=0.1, inplace=True)
Linear (in_features=2000, out_features=256, bias=True)	**Relative Multi-Head Attention 2** (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)
LeakyReLU (negative_slope=0.1, inplace=True)	**Dropout** (p=0.5, inplace=False)
Linear (in_features=256, out_features=6, bias=True)	**LeakyReLU** (negative_slope=0.1, inplace=True)
OUTPUT: Global pose vector	**INTERMEDIATE OUTPUT:** Intermediate relative pose multidimensional vector
INTERMEDIATE INPUT: **Query:** Intermediate global pose multidimensional vector **Key:** Intermediate relative pose multidimensional vector **Value:** Intermediate relative pose multidimensional vector	
Common Multihead Attention (in_features=2000, out_features=2000, bias=True, number_of_heads=8, dropout_p=0.65)	
Dropout (p=0.5, inplace=False)	
LeakyReLU (negative_slope=0.1, inplace=True)	
Linear (in_features=2000, out_features=256, bias=True)	

LeakyReLU (negative_slope=0.1, inplace=True)
Linear (in_features=256, out_features=6, bias=True)
OUTPUT: Relative pose transformation vector

Table B.4 GlobalRelativeTransformerVO architecture details

GlobalRelativeTransformerVO	
Note: This experimental design, strongly inspired by the Transformer model presented in [15], attempted to use an auxiliary global pose output to create an intermediate multidimensional global pose vector that could be used as the Encoder output in [15] which is fed as a secondary input to the Decoder. The idea was to let the TransformerDecoder module learn to model the relation between global pose estimates and relative pose transformation estimates. The hope was that it would use this relationship to tune relative pose transformation estimates according to how well they assembled into global pose estimates. This design does not perform well and was quickly discarded.	
INPUT: Concatenated image pair sequence	
FlowNetSimple Convolutional Encoder	
LSTM (hidden_size = 1000, num_layers=2, bidirectional=True)	
Dropout (p=0.5, inplace=False)	
TransformerEncoder (in_features=2000, heads=8, dropout_probability=0.65, layers=1)	-
INTERMEDIATE OUTPUT: Intermediate global pose multidimensional vector	-
Dropout (p=0.5, inplace=False)	-
LeakyReLU (negative_slope=0.1, inplace=True)	-
Linear (in_features=2000, out_features=256, bias=True)	-
LeakyReLU (negative_slope=0.1, inplace=True)	-
Linear (in_features=256, out_features=6, bias=True)	-
OUTPUT: Global pose vector	-
INTERMEDIATE INPUT: **Input 1:** LSTM Output vector of size 2000 **Input 2:** Intermediate global pose multidimensional vector from the TransformerEncoder	
TransformerDecoder **(in_features=2000, heads=8, dropout_probability=0.65, layers=1)**	
Dropout (p=0.5, inplace=False)	
LeakyReLU (negative_slope=0.1, inplace=True)	
Linear (in_features=2000, out_features=256, bias=True)	
LeakyReLU (negative_slope=0.1, inplace=True)	

Linear (in_features=256, out_features=6, bias=True)
OUTPUT: Relative pose transformation vector

APPENDIX C TESTED LOSSES

BatchSegmentMSELoss:

$$\boldsymbol{Loss_{MSE}} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{j=0}}^{\boldsymbol{N-1}}\left(\sum_{\boldsymbol{i=0}}^{\boldsymbol{n-1}}\mathbf{100} * \|\widehat{\boldsymbol{\varphi}}_{\boldsymbol{i}} - \boldsymbol{\varphi_i}\|_2^2 + \|\widehat{\boldsymbol{t}}_{\boldsymbol{i}} - \boldsymbol{t_i}\|_2^2\right)_{\boldsymbol{j}}$$

This is the candidate loss function presented in Chapter 5.

EulerGlobalRelativeMSELoss & EulerSeparateGlobalRelativeMSELoss:

$$\boldsymbol{Loss_{GlobalRelativeMSE}} = \boldsymbol{w_G Loss_{MSE}(G)} + \boldsymbol{w_R Loss_{MSE}(R)}$$

$\boldsymbol{Loss_{GlobalRelativeMSE}}$ uses a weighted ($\boldsymbol{w_G}$) MSE loss for global pose estimates ($\boldsymbol{G}$) and a weighted ($\boldsymbol{w_R}$) MSE loss for relative pose transformation estimates ($\boldsymbol{R}$). This loss is used twice in the code for *EulerGlobalRelativeMSELoss* and *EulerSeparateGlobalRelativeMSELoss*. Those are the same loss. The former is used on single output networks that generate relative pose transformation estimates that are then assembled into global pose estimates. The latter is used on networks with an auxiliary output estimating global poses.

TemporalGeometricConsistencyLoss:

Given

$$\boldsymbol{P_i} = \boldsymbol{P_{i-1} T_{i-1,i}} \ ,$$

where $\boldsymbol{T_{i-1,i}} \in \mathbb{R}^{\mathbf{4\times4}}$ is a transformation matrix between time step $\boldsymbol{i-1}$ and $\boldsymbol{i}$; and $\boldsymbol{P} \in \mathbb{R}^{\mathbf{4\times4}}$ is a transformation matrix at time step $\boldsymbol{i}$. Then, through rigid body composition

$$\boldsymbol{P_{i,i+4}} = \boldsymbol{P_{i+2,i+4} P_{i,i+2}}^{\mathbf{-1}} \ ,$$

$$\boldsymbol{P_{i,i+2}} = \boldsymbol{P_{i+1,i+2}}\ \boldsymbol{P_{i,i+1}}^{\mathbf{-1}}, \boldsymbol{P_{i+2,i+4}} = \boldsymbol{P_{i+3,i+4}}\ \boldsymbol{P_{i+2,i+3}}^{\mathbf{-1}},$$

for a segment composed of five poses. Six sub-losses, which enforces the rules of rigid body composition, can be computed:

$$\boldsymbol{L_0} = \left\|\widehat{\boldsymbol{P}}_{\boldsymbol{i,i+1}} - \boldsymbol{P_{i,i+1}}\right\|_{\mathbf{2}}^{\mathbf{2}}, \boldsymbol{L_1} = \left\|\widehat{\boldsymbol{P}}_{\boldsymbol{i+1,i+2}} - \boldsymbol{P_{i+1,i+2}}\right\|_{\mathbf{2}}^{\mathbf{2}}, \boldsymbol{L_2} = \left\|\widehat{\boldsymbol{P}}_{\boldsymbol{i+2,i+3}} - \boldsymbol{P_{i+2,i+3}}\right\|_{\mathbf{2}}^{\mathbf{2}}$$

$$\boldsymbol{L_3} = \left\|\widehat{\boldsymbol{P}}_{i+3,i+4} - \boldsymbol{P}_{i+3,i+4}\right\|_2^2,\ \boldsymbol{L_4} = \left\|\widehat{\boldsymbol{P}}_{i,i+2} - \boldsymbol{P}_{i,i+2}\right\|_2^2,\ \boldsymbol{L_5} = \left\|\widehat{\boldsymbol{P}}_{i+2,i+4} - \boldsymbol{P}_{i+2,i+4}\right\|_2^2$$

$$\boldsymbol{L_6} = \left\|\widehat{\boldsymbol{P}}_{i,i+4} - \boldsymbol{P}_{i,i+4}\right\|_2^2.$$

The *TemporalGeometricConsistencyLoss* used in this project is composed of a weighted ($\boldsymbol{w_{TGC}}$) sum of the sub-losses with a weighted ($\boldsymbol{w_{MSE}}$) MSE loss. This loss is directly inspired by [43].

$$\boldsymbol{Loss_{TemporalGeometricConsistency}} = \boldsymbol{w_{TGC}}\left(\frac{\boldsymbol{1}}{\boldsymbol{N}}\sum_{j=0}^{N-1}\left(\sum_{i=0}^{6}\boldsymbol{L_i}\right)_j\right) + \boldsymbol{w_{MSE}}\boldsymbol{Loss_{MSE}}$$

<u>LoopedBatchSegmentMSELoss:</u>

Given

$$\boldsymbol{P_i} = \boldsymbol{P_{i-1}}\boldsymbol{T}_{i-1,i},$$

and

$$\boldsymbol{P_i} = \begin{bmatrix}\boldsymbol{R}_i & \boldsymbol{t}_i \\ \boldsymbol{0} & \boldsymbol{1}\end{bmatrix},$$

where $\boldsymbol{T}_{i-1,i} \in \mathbb{R}^{4\times4}$ is a transformation matrix between time step $\boldsymbol{i-1}$ and $\boldsymbol{i}$, $\boldsymbol{P_i} \in \mathbb{R}^{4\times4}$ is a transformation matrix at time step $\boldsymbol{i}$ composed of rotation matrix $\boldsymbol{R_i} \in \boldsymbol{SO(3)}$ and location vector $\boldsymbol{t_i} \in \mathbb{R}^{3\times1}$.

Then a matrix $\mathbf{Q}_{0\to n-1} \in \mathbb{R}^{3\times n}$ of locations from time step $\boldsymbol{0}$ to $\boldsymbol{n-1}$ can be written as:

$$\mathbf{Q}_{0\to n-1} = [\mathbf{t}_0 \ldots \mathbf{t}_{n-1}].$$

A similarity matrix describing the distance between two locations on the trajectory can be computed using the following algorithm (pseudo-code):

```
1  for $\boldsymbol{t_i}$ in $\mathbf{Q}_{0\to n-1}$:
2      for $\boldsymbol{t_j}$ in $\mathbf{Q}_{0\to n-1}$:
3          $\boldsymbol{d_{i,j}} = \left\|\boldsymbol{t_i} - \boldsymbol{t_j}\right\|_2$
```

Then a matrix $\boldsymbol{D} \in \mathbb{R}^{n\times n\times 3}$ of ground truth locations similarity $\boldsymbol{d_{i,j}} \in \mathbb{R}^{3\times1}$ can be written as:

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{d_{0,0}} & \cdots & \boldsymbol{d_{0,n-1}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{d_{n-1,0}} & \cdots & \boldsymbol{d_{n-1,n-1}} \end{bmatrix},$$

and a matrix $\boldsymbol{D} \in \mathbb{R}^{\boldsymbol{n \times n \times 3}}$ of estimated locations similarity $\boldsymbol{\hat{d}_{i,j}} \in \mathbb{R}^{\boldsymbol{3 \times 1}}$ can be written as:

$$\boldsymbol{\hat{D}} = \begin{bmatrix} \boldsymbol{\hat{d}_{0,0}} & \cdots & \boldsymbol{\hat{d}_{0,n-1}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\hat{d}_{n-1,0}} & \cdots & \boldsymbol{\hat{d}_{n-1,n-1}} \end{bmatrix}.$$

From the similarity matrix of the ground truth locations $\boldsymbol{D}$, a Boolean mask identifying the locations that are within a specified radius (the loops) can be computed using:

$$\boldsymbol{M_{Loops}} = \boldsymbol{D} \leq \boldsymbol{radius}.$$

By applying this Boolean mask on the similarity matrix $\boldsymbol{\hat{D}}$ and $\boldsymbol{D}$, only the distances of the identified loops are kept, everything else is set to zero. The mean squared error between the estimated loop distances and ground truth loop distances can then be computed using

$$\boldsymbol{Loss_{loop}} = \left\| \boldsymbol{M_{Loops}} \circ \boldsymbol{\hat{D}} - \boldsymbol{M_{Loops}} \circ \boldsymbol{D} \right\|_{\boldsymbol{2}}^{\boldsymbol{2}},$$

where ∘ is the Hadamard product (element-wise product).

For this project, *LoopedBatchSegmentMSELoss* combines a weighted $\boldsymbol{Loss_{loop}}$ and a weighted $\boldsymbol{Loss_{MSE}}$:

$$\boldsymbol{Loss_{LoopedMSE}} = \boldsymbol{w_{loop}} \boldsymbol{Loss_{loop}} + \boldsymbol{w_{MSE}} \boldsymbol{Loss_{MSE}}.$$

APPENDIX D TESTED MODEL-LOSS COMBINATIONS

Table D.1 Tested model-loss combinations

Model	Loss	Mean Validation MSE	Note(s)
SelfAttentionVO	BatchSegmentMSELoss	0.00142	Candidate Model and Loss
SelfAttentionVO	EulerGlobalRelativeMSELoss	0.00160	
SelfAttentionVO	TemporalGeometricConsistencyLoss	0.00145	Discarded. Qualitative visual analysis of the reconstructed trajectories (KITTI 07, 10) showed poor accuracy.
SelfAttentionVO	LoopedBatchSegmentMSELoss	0.00193	
GlobalRelative SelfAttentionVO	EulerSeparateGlobalRelativeMSELoss	0.00140	Discarded. Qualitative visual analysis of the reconstructed trajectories (KITTI 07, 10) showed poor accuracy.
GlobalRelative SelfAttentionVO	GlobalRelative TemporalGeometricConsistencyLoss	0.00128	Discarded. Qualitative visual analysis of the reconstructed trajectories (KITTI 07, 10) showed poor accuracy.
GlobalRelative TransformerVO	EulerSeparateGlobalRelativeMSELoss	0.00188	
GlobalRelative TransformerVO	GlobalRelative TemporalGeometricConsistencyLoss	0.00161	
Simple SelfAttentionVO	BatchSegmentMSELoss	0.00314	