Alma Mater Studiorum - University of Bologna

Computer Science and Engineering - DISI

Artificial Intelligence

A study on tackling visual odometry by a transformer architecture

Master degree thesis

Supervisor

Prof. Luigi Di Stefano

Co-supervisor

Luca De Luigi

Candidate

Xiaowei Wen

Academic Year 2021-2022- Second session

Dedicated to my parents.

Summary

This dissertation describes a deepening study about Visual Odometry problem tackled with transformer architectures. The objectives were: create a synthetic dataset using BlenderProc2 framework, try different kind of transformer architectures which includes: ResNet feature-extractor with encoder and a small MLP, ResNet feature-extractor with encoder-decoder and a MLP, ResNet-feature extractor with encoder-decoder and pose Auto-encoder.

"Dio benedica quelle persone che quando incroci il loro sguardo per sbaglio, sorridono."

Thanks

First, I would like to express my deepest gratitude to Professor Luigi Di Stefano and Luca De Luigi for the guidance and support during the internship

Second, I would like to thank my parents for their moral support and for their patience.

Third, I would like to thank my girlfriend and friends for being patient with me and to put up with my complaining.

Bologna, 06 October 2022 Xiaowei Wen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this section we will present the summarized content of the whole thesis.

1.1 Background

Computer vision (CV) is a field of artificial intelligence that deals with the study of how computers can be made to gain high-level understanding from digital images or videos. If AI allows the computer to think like a human, computer vision allows the computer to see like a human.

CV works like the human visual system, with the big difference in the fact that human uses year and year of experience to help the mind to understand what it is seeing. As the biological neurons processes the information in the brain, the artificial neurons processes the information in the artificial neural network following the *Hebbian Plasticity* 1949 rule: the connection between two neurons is strengthened if they are active at the same time.

In recent years, the deep learning has revolutionized the CV field, achieving excellent results in many tasks, like: image classification, object detection, semantic segmentation, image captioning, image generation, etc.

The image classification task consists of assigning a label to an image with only one object (Figure 1.1).

Figure 1.1: Image classification: this image is classified as a tulip

The object detection tasks consists of assigning a label and a bounding box to each object in the image. The bounding box is a rectangle that encloses the object(Figure 1.2).

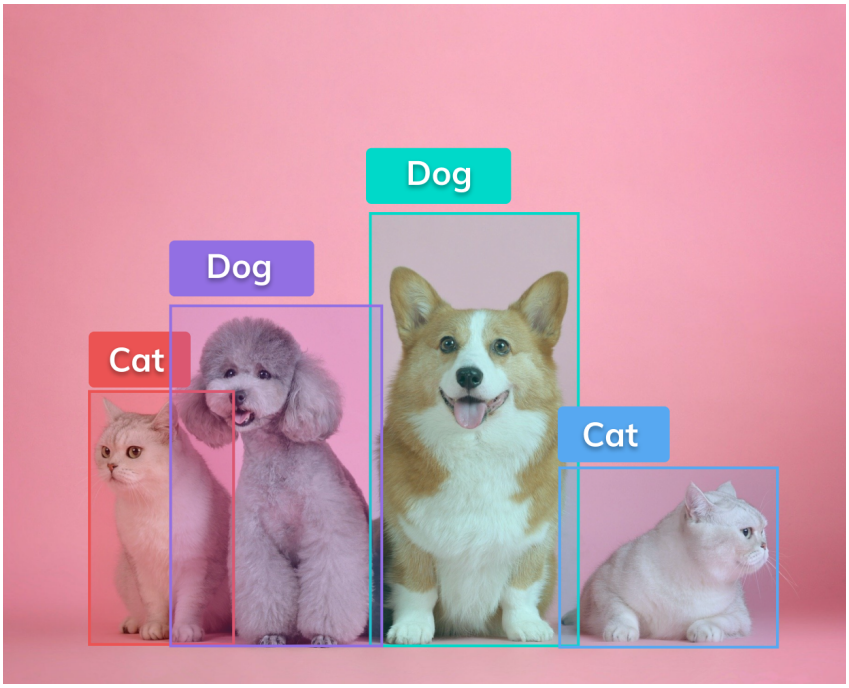

Figure 1.2: Object detection: this image contains two classes of objects, cat and dog.

The semantic segmentation task consists of assigning a label to each pixel of the image(Figure 1.3).

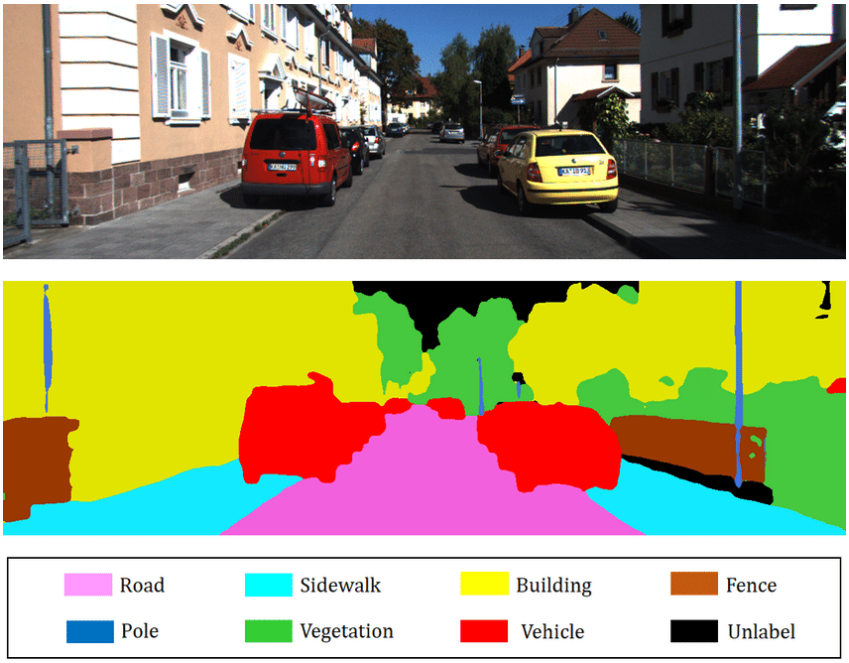

Figure 1.3: Semantic segmentation: each pixel is assigned a label.

Then, the modern CV systems can be used not only on the images, but also on video, like surveillance cameras to perform the real-time object detection and tracking, the most famous model is YOLOv3 Redmon and Farhadi 2018 (Figure 1.4).

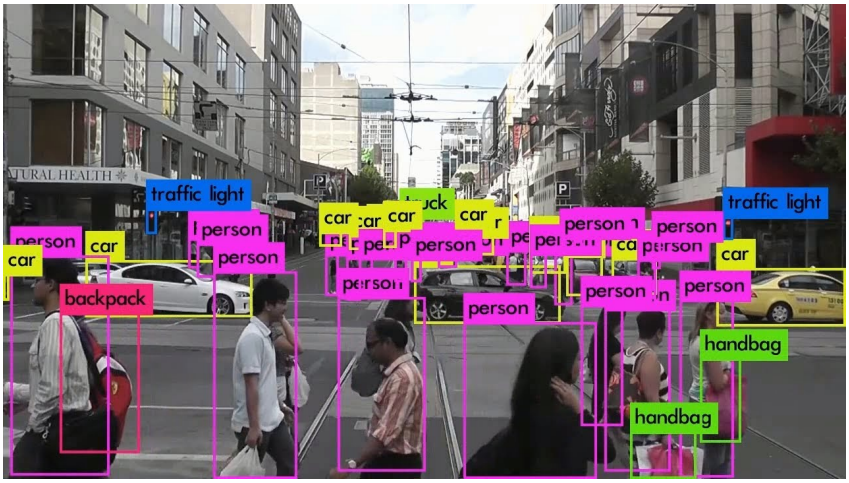

Figure 1.4: YOLO-V3 in action.

1.2 Problem

The term "odometry" originated from two Greek works *hodos* (meaning "journery" or "travel") and *metron* (meaning "measure"). This derivation is related to the estimation of the change in a robot's pose (translation and rotation) over time. Mobile robot use data from motion sensor to estimate their position relative to their initial location, this is called odometry. VO is a technique used to localize a robot by using only a stream of images acquired from a single or multiple camera. There are different ways to classify the typology of Visual Odometry:

- based on the camera setup:
  - Monocular VO: using only one camera;
  - Stereo VO: using two cameras;
- based on the information:
  - Feature based method: which extracts the image feature points and tracks them in the image sequence;
  - Direct method: a novel method which uses the pixel intensity in the image sequence directly as visual input.
  - Hybrid method: which combines the two methods.
- Visual inertial odometry: if a Inertial measurement unit (IMU) is used within the VO system, it is commonly referred to as Visual inertial odometry.

We can represent the pose in different ways, for example: **euler angles**, **quaternions**, **SO(3)**, **rotation matrices** combined with **translation vectors**.

The goal is to create a Neural network (NN), using a **ResNet** to extract features from images and the **transformer** presented by Vaswani et al. 2017, which is able to estimate a sequence of camera's pose given a sequence of images.

1.3 Why Transformer?

We think that the transformer is a good candidate to solve the problem of visual odometry because it is able to learn the sequence of images and the sequence of poses in a self-supervised way.

Although, the transformer contrasts strongly with CNNs. Because in CNNs the features are statically weighted using pretrained weights, while in the transformer

the features are dynamically weighted based on the context and receptive fields of individual network layers are typically local and limited by the convolutional kernel size.

The success of the CNN derives from the fact the shared weights explicitly encode how specific identical patterns are repeated in images, this ensures the convergence also in relatively small dataset, but also limits the modelling capacity. Meanwhile, the vision transformers do not enforce such strict bias. But in the same time, transformer has the higher learning capacity, but it's harder to train.

So, given the high learning capacity of the transformer, and its capability to adapt to various tasks also for the fact that the transformer is a general purpose architecture, we think that its is a good candidate to solve the problem of visual odometry.

1.4 Solution

We tried to tackle the problem by designing a deep neural network which is composed by a feature extractor, the transformer and a MLP to predict the pose. We feed the feature extractor with a sequence of images, we tried both grey-scale and RGB images, in this way, we obtain a sequence of embeddings (both size 512 and 2048), the embedding are then fed into the transformer (both encoder and encoder-decoder version) and the output of the transformer is fed into the MLP to predict the pose.

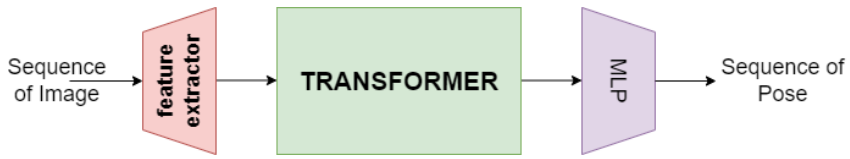

Figure 1.5: General representation of the model.

We use a sequence of image because the transformer model, originally designed for the machine translation, it requires as input a sequence of embeddings, then it outputs another sequence of embeddings.

1.5 Thesis Organization

First chapter introduces the general content about thesis and gives a short presentation of the topic, the problem and the solution we propose;

Second chapter a deepening about the theoretical foundations used during the stage and the project;

Third chapter presents the datasets used during for the training and the testing of the model;

Fourth chapter presents the experiments did during to develop the system;

Fifth chapter presents the different implementations of the system;

Sixth chapter discusses about the results and possible future developments.

During the drafting of the essay, following typography conventions are considered:

- the acronyms, abbreviations, ambiguous terms or terms not in common use are defined in the glossary, in the end of the present document;
- the first occurrences of the terms in the glossary are highlighted like this: word;
- the terms from the foreign language or jargon are highlighted like this: *italics*.

Chapter 2

Theoretical foundations

In this chapter we will present the theoretical knowledge useful to understand the content from successive chapters.

2.1 Deep Learning

Deep learning method is part of machine learning methods based on artificial neural network with representation learning. The learning process can be supervised, semi-supervised, or unsupervised.

There is a very large variety of deep learning architectures, some of them are specialized in some fields meanwhile others have a broader usage, especially, there are CNNs and Transformers.

In recent years, the field of computer vision has been growing in complexity and the number of applications has been increasing, in addition to those presented in Section 1.1 Computer vision, there are Simultaneous Localization and Mapping (SLAM) and visual odometry which is a task in which the robot is able to understand where it is and how it is oriented.

The development of computer vision has been a long process, the growth is favoured by the development of new hardware components and new challenges, about the latters, we have CIFAR-10 (Doon, Kumar Rawat, and Gautam 2018), Fashion-MNIST(Xiao, Rasul, and Vollgraf 2017), MS-Coco (Lin et al. 2014) and ImageNet (Deng et al. 2009). These datasets are often used as benchmark for novel models.

For the architectures, starting from AlexNet (Krizhevsky, Sutskever, and Hinton 2012), then VGG (Simonyan and Zisserman 2014), Inception-V1 (Szegedy, Liu, et al.

2014), Inception-V2(Szegedy et al. Szegedy, Vanhoucke, et al. 2015), ResNet (He et al. 2015), etc., the complexity of the models has increased enormously. Each of these models introduced some innovations and improved the performance on the benchmarks, for example:

- AlexNet introduced the concept of the *convolutional neural network* (CNN) and use of the separation of the models into two different GPUs.
- VGG introduced the concept of stage, which repeated more times, composes the model.
- Inception-V1, Inception-V2 and Inception-V3 which are based on the concept of *inception module* which was composed by different paths that the input has to go through to reach the output.

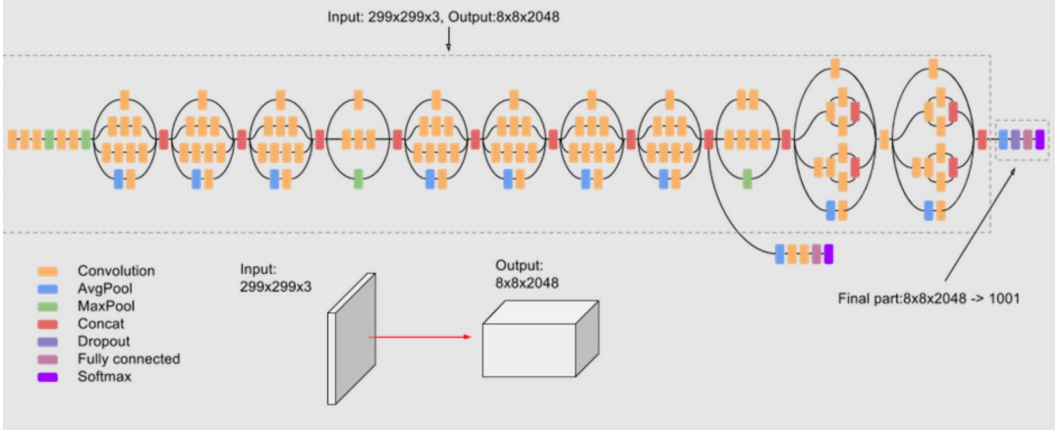

Figure 2.1: Inception V3 Structure.

- ResNet is a model that is based on the concept of *residual network* which is composed by several blocks of the same type with the skip connections:

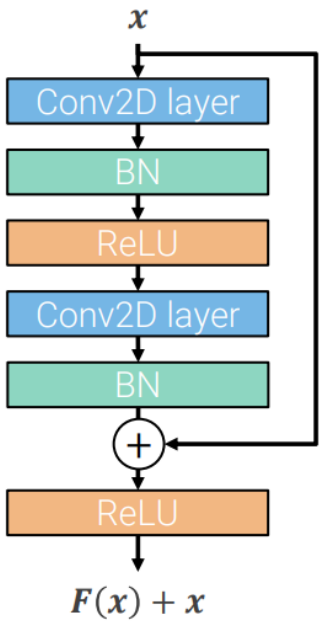

Figure 2.2: Skip connection.

> Basically, the input of the block is added to the output before feeding it to the next block, in this way, we can avoid the vanishing gradient problem making easier the training process.

After this, the computer-visionists lend the Transformer architecture (Vaswani et al. 2017) from Natural Language Processing (NLP), bringing up ViT (Dosovitskiy et al. 2020) which is based on the Multi-Head Attention (MHA) mechanism. A multi-head attention is a module of attention mechanisms repeated several times in parallel. In this way, the model can attend to different parts of the input, forming, in this way, the cross-attention over different parts of the input. For major details, please refer to §2.1.2 Transformer.

2.1.1 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a class of artificial neural network, it is used in almost every imagery related task, such as image classification, object detection, image segmentation, etc.

The CNN take an input image, assign importance (learnable weights and biases) and process the input image by using the convolution operation extracting features. There are two important parameter in the convolution operation, the kernel size and the stride. The kernel is a matrix which is used to perform the convolution operation, the stride is the number of pixels the kernel slides each step over the

input image to produce a new pixel of the output feature map. With stride, we can control the size of the output image, if the stride is equal to 1, the output image will have the same size of the input image, if the stride is equal to 2, the output image will have half the size of the input image.

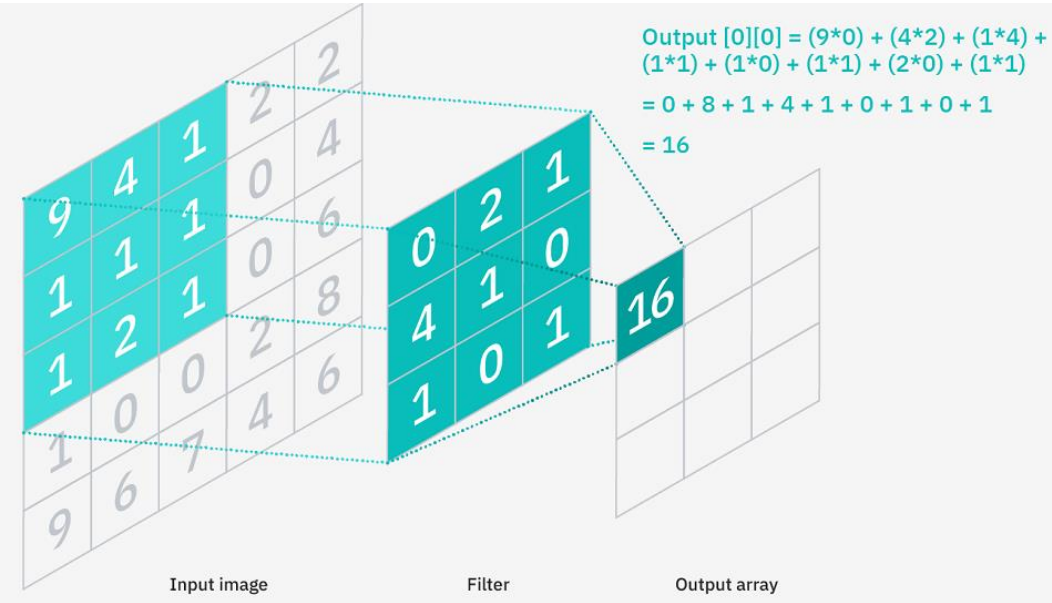

Figure 2.3: Convolutions: every single element of the output feature map is obtained by summing the element-wise product between the elements from the input feature map and the kernel. The whole feature map is then obtained sliding the kernel over the input feature map.

Then, there are pooling layers, usually max-pooling and average pooling, which can reduce the dimensionality of the feature maps by setting strides $>= 2$, which is useful to reduce the computational cost. For example, max-pooling is computed as showed in the image:

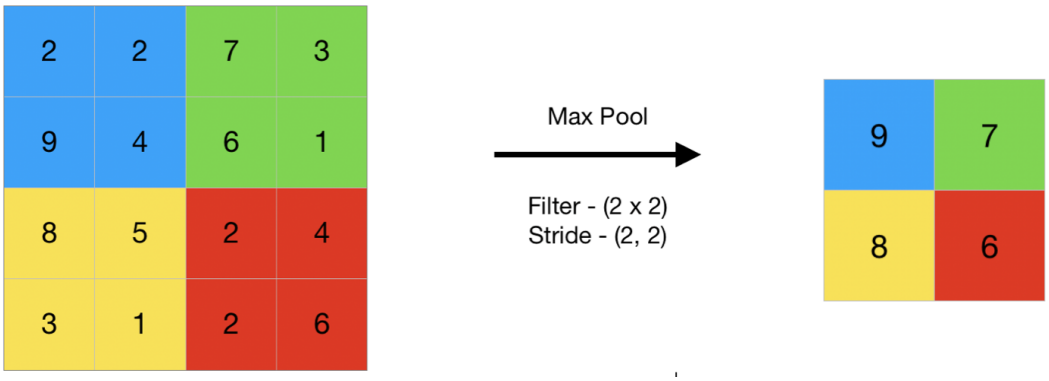

Figure 2.4: Max-pooling: essentially, it strides over the input image and takes the max value of the area covered by the kernel.

With stride = 2, sliding over the input feature map and taking the maximum value of the window, the dimensionality of the feature map is reduced. Another important component is the activation function, Rectified Linear Unit (ReLU) is the most used one, it is defined as:

$$ReLU(x) = \max(0, x) \tag{2.1}$$

Which guarantees the non-linearity of the network, allowing the network to learn more complex features. These are the main components of a CNN, but there are other components, such as batch normalization and dropout which are used to improve the performance of the network reducing the over-fitting. Increasing the number of layers and combining the pooling layers, the CNN is able to extract more and more complex features, such as edges, lines, shapes, etc. Currently, the most used CNN architecture is the ResNet which will be used in the project as feature-extractor.

2.1.2 Transformer

The transformer architecture is a class of neural network architecture, born for the task of machine translation, but it has been used in many vision tasks.

As introduced in Vaswani et al. 2017, the Transformer is a model architecture based entirely on attention mechanism. The first step of attention mechanism is to compute the Q, K and V vectors, by multiplying the input vector x by the weight matrices W_q, W_k and W_v. Then, the attention weights are computed by using the scaled dot product attention, which is the softmax of the dot product between the query and the key vectors divided by the square root of the dimensionality of the key vector. Finally, the attention weights are multiplied by the value vector to

obtain the output vector. The output vector is then passed through a feed-forward neural network, which is composed by two linear layers with ReLU activation function, added to the input vector and normalized by the layer normalization. The self-attention module is then repeated N times, where N is the number of layers.

The whole process can be summarized as the:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.2}$$

And the graphical representation is:

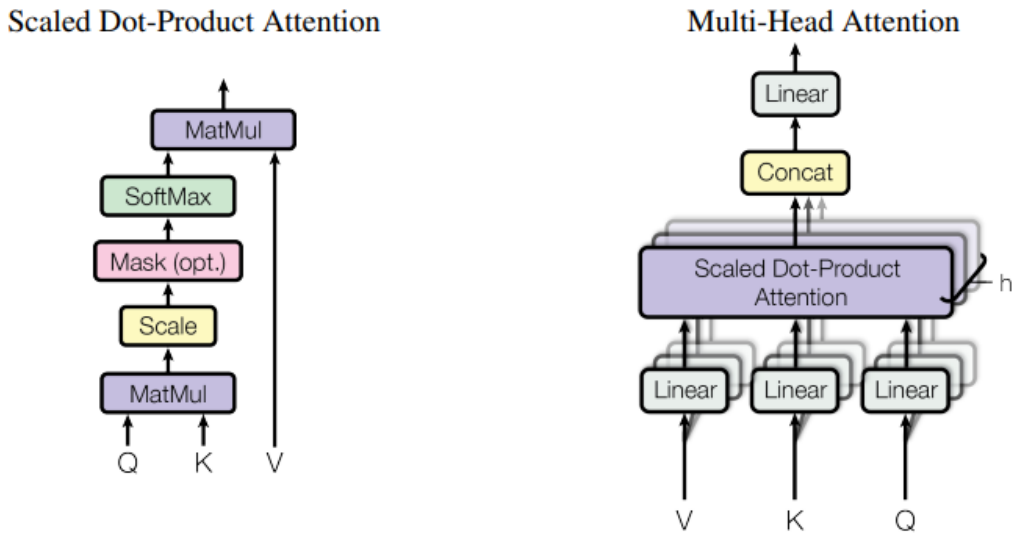

Figure 2.5: Attention mechanism: (Left) scaled-dot-product. (right) Multi-head attention which is obtained by combining many scaled-dot-product attention.

Another important notion introduced is the multi-head attention, which is obtained by using more scaled dot product attention, each one with different weights, and concatenating each output vectors. Then, using a slightly modified version of the self-attention module, the encoder module is used as decoder, which takes as input also the output sequence, and repeating the number of encoder and decoder modules, we obtain the whole transformer architecture, as follows:

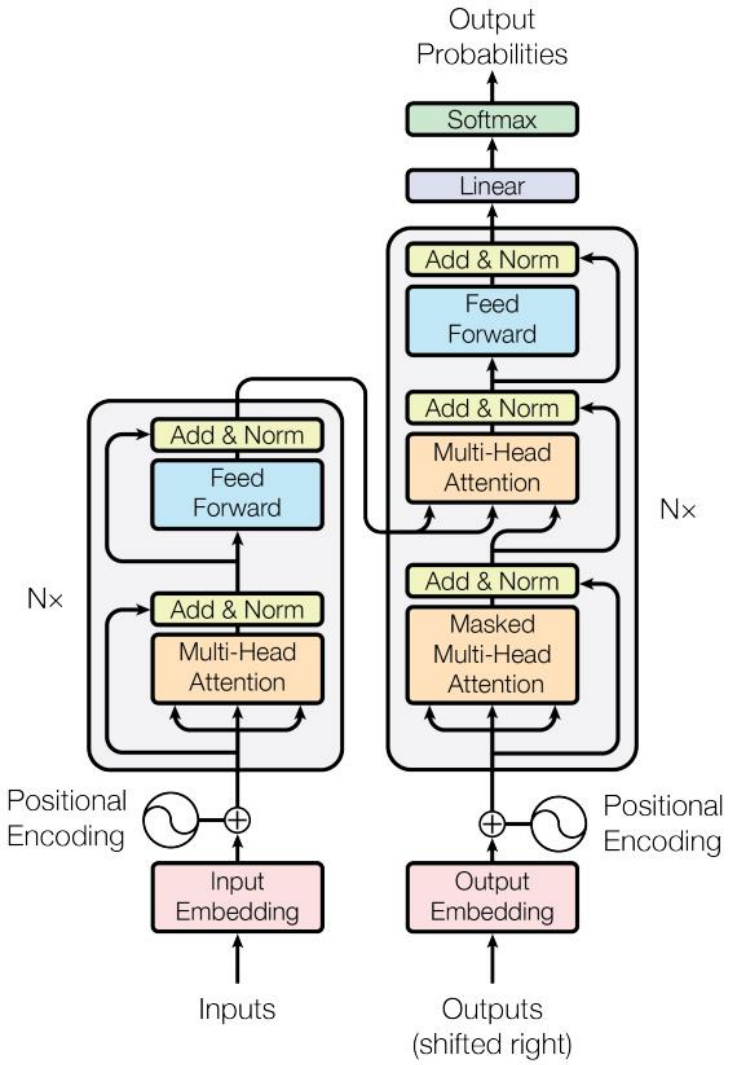

Figure 2.6: Transformer architecture: the encoder and decoder modules are composed by a self-attention module and a feed-forward neural network repeated N times.

With this architecture the new state-of-the-art results have been achieved in Natural Language Processing, especially in machine translation. Then the adapted version applied to vision tasks also brought very good results, such as in object detection, image captioning, etc.

2.2 Odometry

As introduced in §1.2, the problem of odometry is about the estimation of the change in a robot's pose over time.

The odometry, also known as self-localization, can be classified in different ways,

in §2.2.1 there is a more detailed description of the different types of odometry.

2.2.1 Taxonomy

There different types of odometry, which based on the classification of Alkendi, Seneviratne, and Zweiri 2021 can be divided into two main categories: *GNSS available* and *GNSS not available*.

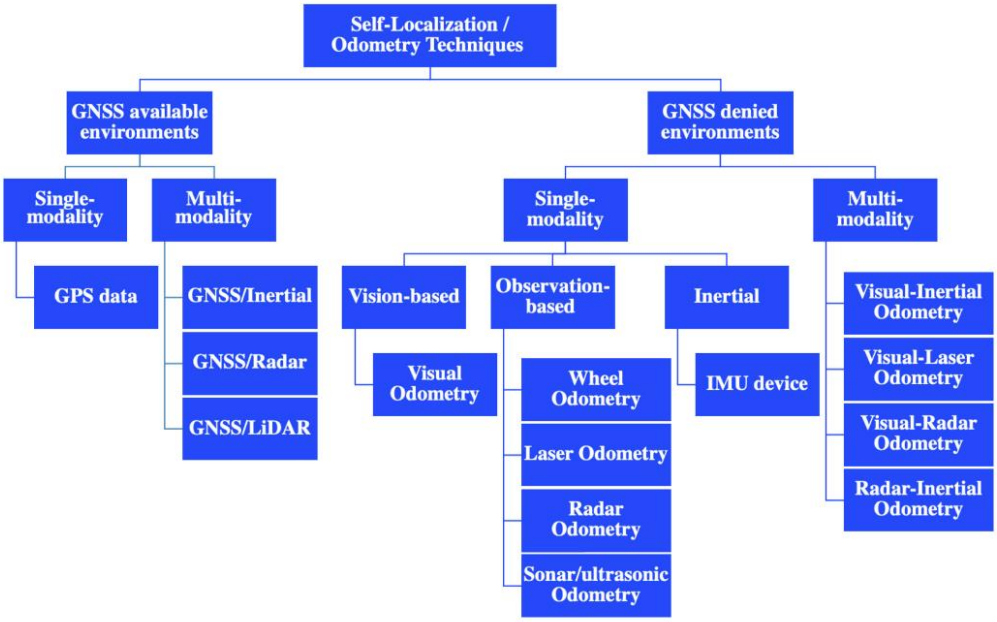

Figure 2.7: Taxonomy of odometry techniques

2.2.2 Reference systems

To tackle the problem of odometry, we need as to choose the representation system to adopt. There are many way of representing the pose of the camera or the robot, but the most common are the *Euler angles* and the *quaternions* and *rotation matrix* combined with *translation matrix*.

2.2.3 State of the art

2.3 Literature protocol

In the literature, when we need to develop a new project, there is a protocol which should be followed to increase the possibility of success. This protocol is composed by a sequence of steps, the success of every step is fundamental for the

continue of the next steps. The steps are:

1. Study of the state of the art and deepening about the other projects' results.
2. Seeking for the dataset, we should look for a dataset which fits to our purpose, we should understand the characteristics of the datasets.
3. We should find some projects in oder to use them for the comparison
4. Validate the dataset using the other models, trying to reach the same results as the authors'.
5. Build the model and use it as baseline.
6. Over-fit the model with a single prediction target class, in our case a single sequence to verify the network capacity.
7. Over-fit the model with two and more prediction target classes, in this way, we are verifying that the model can learn more than one target, which is useful for us to understand which is the limit of the network in term of capacity.
8. Train the model with the whole dataset, trying to improve the results achieved by the baseline, by changing the hyper-parameters or by changing the model.
9. Fine-tuning, perform a fine tuning of the neural network can squeeze the last drops of performance of the network.
10. Compare the results with the state of the art, discussion about the results and the possible improvements.

Chapter 3

Datasets

In this chapter we will present the datasets created and used for the visual odometry.

3.1 Kitti

The odometry benchmark consists of 22 stereo sequences, saved in loss less png format: 11 sequences are provided with ground truth trajectories for training and 11 sequences (11-21) without ground truth trajectories for evaluation.

This odometry benchmark is a subset of KITTI Vision Benchmark suite Geiger, Lenz, and Urtasun 2012b.

3.1.1 Scene

The images represents a various of scenes from mid-sized city, rural areas and on highways.

Figure 3.1: KITTI - example of scene

3.1.2 Image generation

Each sequence of the KITTI dataset is composed of by four sequences of images: left-coloured, right-coloured, left-grey and right-grey. Each one is captured by a camera mounted on the top of vehicle. They calibrated the four video cameras intrinsically and extrinsically and rectified the input image. Then they computed the 3D rigid motion parameters which relate the coordinate system of the laser scanner.

Meanwhile, the ground-truth is directly given by the output of the GPS/IMU localization unit projected into the coordinate system of the left camera after rectification.

3.1.3 Dataset statistics

The dataset consists of 22 stereo sequences, with a total length of 39.2 km, which was the longest in the time of the publication of the paper. In the dataset, there are no specifically indicate which sequence is used for training, validation or testing, but in this work, the dataset is split as this:

Sets	**N. of Sequence**	**N. Image**
Training set	8	20.098
Validation set	2	1.902
Test set	1	1.201
Total	11	23.201

Table 3.1: KITTI - dataset statistics

The images dimensions about 1240x370 are slightly different, generally varying for few pixels.

3.1.4 Usage

This dataset is the one mainly used, as it is the one of the most famous and most used in the literature.

The sequence **3** and **7** are used for evaluation and testing, because they are the easier ones.

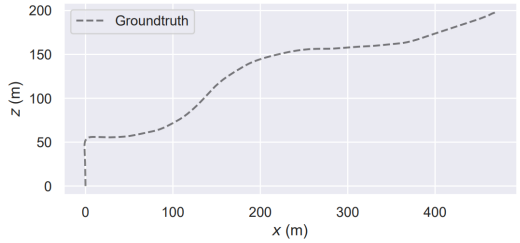

Figure 3.2: KITTI - sequence 3

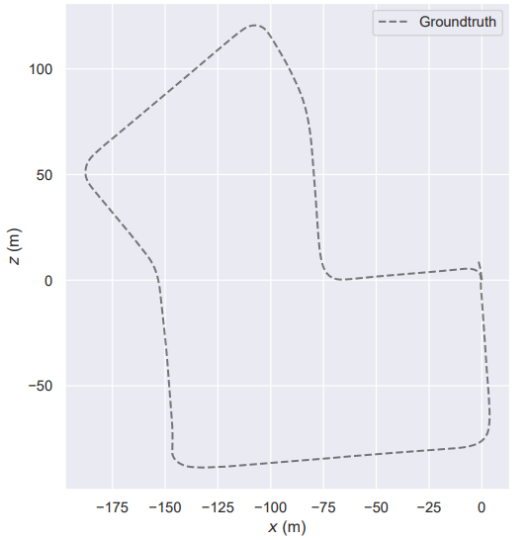

Figure 3.3: KITTI - sequence 7

Initially, to test the model's capacity, the model was trained and evaluated on the same sequence, to see if it's able to reproduce the ground truth. Then, the model was fed with more complex sequences.

3.2 Synthetic

As in there are few real-life datasets for visual odometry, we decided to create a synthetic dataset by using BlenderProc2 framework, which is a procedural photo-realistic rendering framework, and it allows to:

- **Loading**: **.obj*, **.ply*, **.blend*, *BOP*, *ShapeNet* etc.
- **Objects**: set or sample objects poses, apply physics and collision checking.
- **Materials**: set or sample physically-based materials and textures.
- **Lighting**: set or sample lights, automatic lighting of 3D-front scenes.

- **Cameras**: set, sample or load camera poses from file.
- **Rendering**: RGB, stereo, depth, normal and segmentation images/sequences.
- **Writing**: *.hdf5 containers, *COCO* and *BOP* annotations.

3.2.1 Scene

To create the synthetic dataset, the first thing is to create a scene with customized objects, material and textures.

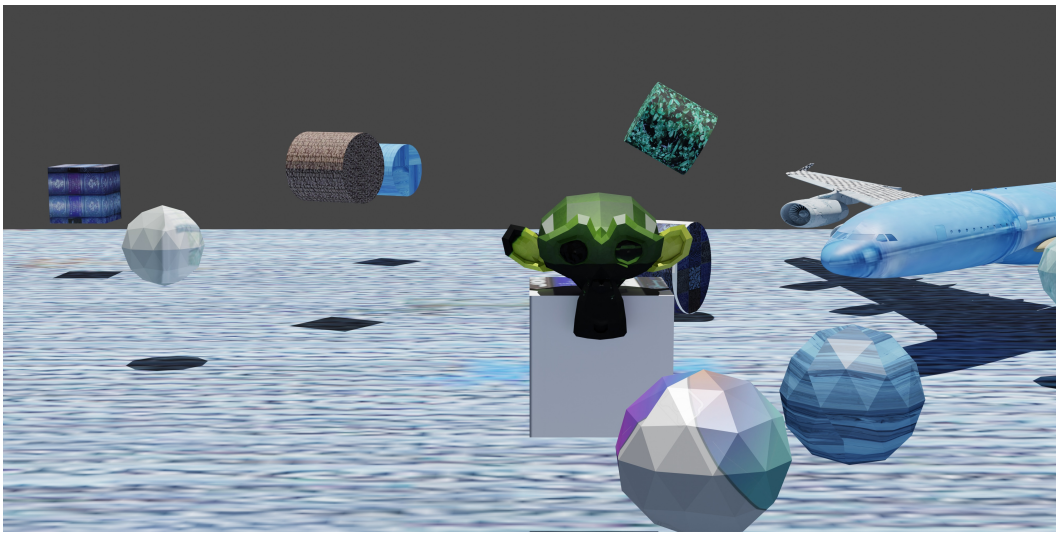

Figure 3.4: Example of scene

The scene is composed by a set of objects, more precisely:

- **a monkey**: which is at the centre of the scene over a cube.
- **a plane**: which is at left corner of the scene.
- **a set of cubes and spheres**: which are placed randomly in the scene.

When rendering the scene, the textures are loaded *randomly*, in the way that in different sequences the textures are different.

3.2.2 Image generation

To generate the sequences, we need to choose the camera position in the scene to do so, we choose randomly a position sampler from the following set for each new pose:

- **disk**: samples a point on a circle or on a 2-ball or on an arc/sector with an inner angle less or equal to 180 degrees.
- **sphere**: samples a point from the surface or from the interior of a solid sphere.
- **part-sphere**: samples a point from the surface or from the interior of a solid sphere which is split in two parts.
- **shell**: samples a point from the volume between two spheres (with radius of the spheres given as parameters).

once we have the next position of the camera, we compute the rotation matrix to be applied to the camera in the way that the camera is always looking at the POI (Point Of Interest) which corresponds to the centre of the scene.

```
rotation = bproc.camera.rotation_from_forward_vec(poi - new_
   position)
```

Code 3.1: Computes the rotatition matrix for the camera.

Then, we apply the rotation matrix to the camera and we generate the image, and by setting a certain number of frames between two poses, the framework renders a sequence of images with relative intermediate poses.

But sometime, it happens that the new camera pose is too close to an object of the scene, so we set two conditions that need to be satisfied, otherwise the sampled camera pose is skipped. The first condition checks if there are obstacles in front of the camera which are too far or too close based on the given *proximity_ checks*, while the second evaluates the interestingness or coverage of the scene.

```
def check_pose(c2w_m, special_obj, bvh_tree):
    if not bproc.camera.perform_obstacle_in_view_check(c2w_m,
       {"min": 5.0}, bvh_tree):
        return False
    if bproc.camera.scene_coverage_score(c2w_m, special_
       objects=special_obj) < 0.7:
        return False
    return True
```

Code 3.2: Checks whether the camera pose satisfies the conditions.

But when the new position is too far away from the old position, the rotation of the camera assumes a wrong value during the transition, because it rotates counter-clockwise instead of clockwise, or vice-versa. For example: If we sample the camera

position from a disk at 0, 90, 180, 270 degrees, the rotations should be as in the figure 3.1:

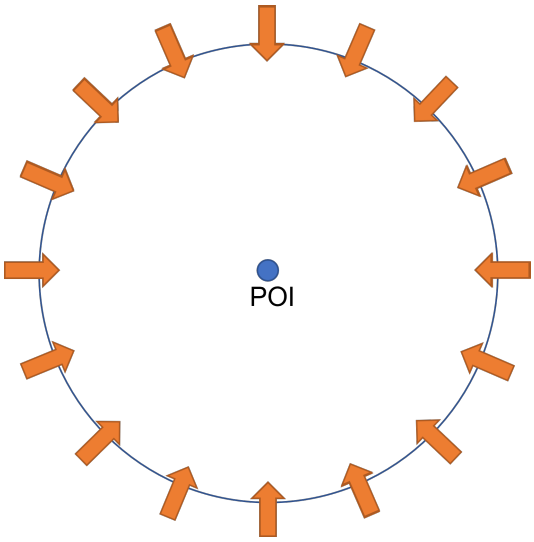

Figure 3.5: Correct transition on the disk

But, the transition from 180 to 270 degrees we obtain is a wrong rotation which is like in figure 3.2:

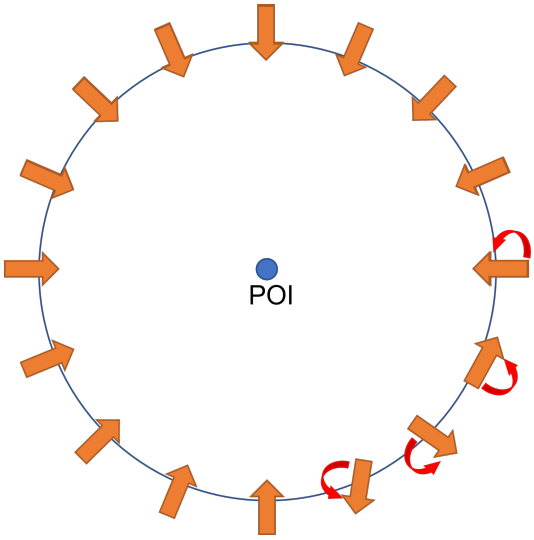

Figure 3.6: Wrong transition on the disk

To solve this problem, we tried different solutions: as first, we sample the next pose near to the previous one, but this solution sometime still fails. The final solution was to sample the new position as previously defined with samplers, but instead of letting the framework to compute the intermediate poses, we manually interpolate

them, and by setting frame number to one, we obtain a sequence of correctly rotating images.

3.2.3 Dataset statistics

In total, we have generated 14 sequences, which are divided as follow:

Sets	N. of Sequence	N. Image
Training set	12	29.100
Validation set	1	1.002
Test set	1	1.003
Total	14	31.105

Table 3.2: Synthetic dataset statistics

Each image has dimension of 1024x308 pixels with 3 RGB channels. The whole dateset has dimension **1.69 GB**.

3.2.4 Usage

By using the dataset at training time the loss function is highly variable reaching values of **thousands**, also because the **Kitti** dataset is much fluid as the trajectory and the camera rotation angles are very small, so, the sequences generated are not similar to the real dataset.

Chapter 4

Experiments

In this chapter we will discuss about different models and different prediction strategies.

4.1 Models

We designed different models, each one with a different architecture. These models will be used to test the different prediction strategies.

4.1.1 Encoder-only model

With only-encoder, we mean that the network has only the encoder part, the decoder part is not present.

We tried to use the encoder part of the network to predict the pose of the camera, using both ResNet18 and ResNet50 as feature extractor. The main difference of ResNet18 and ResNet50 is the number of the output embedding, in fact, ResNet18 has output embedding dimension of 512, while ResNet50 has output embedding dimension of 2048. We also tried with different depth of the network, depth of **6** and **12** layers of encoder. Then, to obtain prediction, we used a fully connected layer to reduce the dimension of the embedding to the desired number of neuron, depending on the prediction strategy. In the figure 4.1 the model architecture is shown.

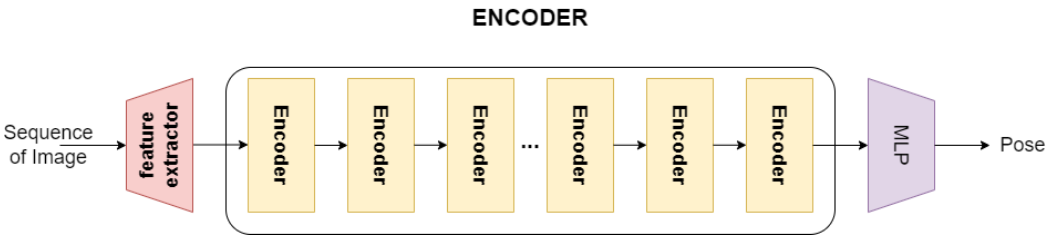

Figure 4.1: Encoder-only transformer

4.1.2 Encoder-decoder

For the encoder-decoder, we used the same encoder of the previous section, and we added a decoder part, which also takes in input a vector parameter as *memory* of the network. In this version of the network, we tried the same configurations of the encoder-only model but the feature extractor ResNet50, because the network requires more memory than those available on GPU. The structure of the network is shown in figure 4.2.

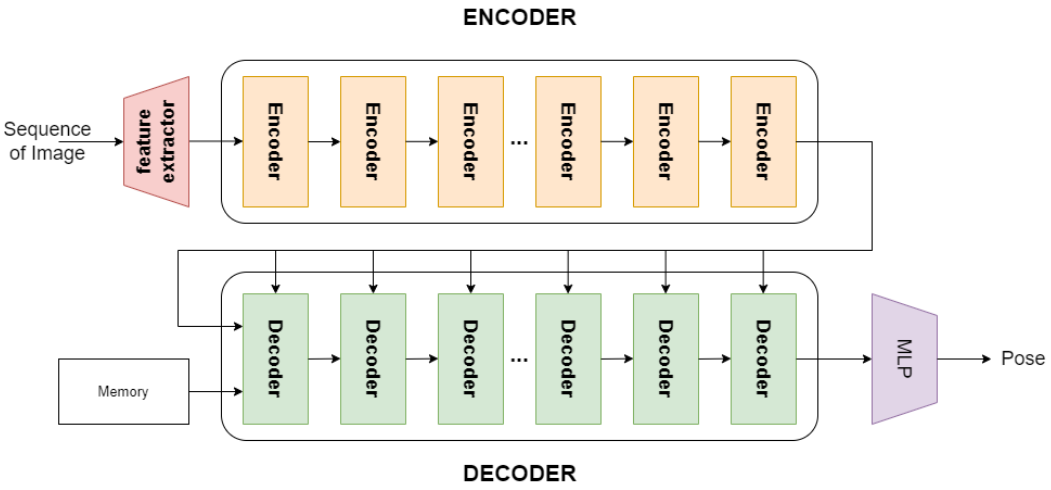

Figure 4.2: Encoder-Decoder transformer

4.1.3 Encoder-Decoder with Auto-encoder

In this version of the model, we used the same encoder-decoder model of the previous section, but we added a pose auto-encoder, which given the ground-truth of the pose, it increases the dimensionality of the input to 512 or 2048 and back to the original dimensionality. The auto-encoder is split into two parts: first part brings the dimensionality from x to 512 and 2048, the second part brings the dimensionality back to x, with x depending on the representation format (6 for euler angles, and 12

for roto-translation matrix).

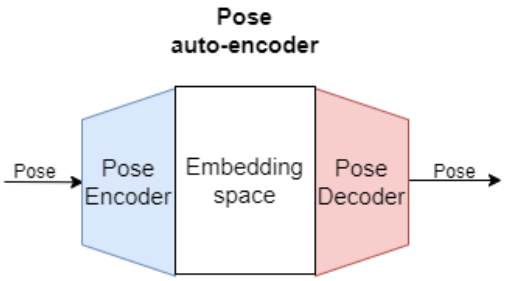

Figure 4.3: Pose auto-encoder

In the Figure 4.3, we can see how the auto-encoder is split in two parts.

We used the first part to create the ground-truth of the pose, the second part to get the prediction of the pose from 512 or 2048 dimensionality. So, the final structure of the model is shown in Figure 4.4.

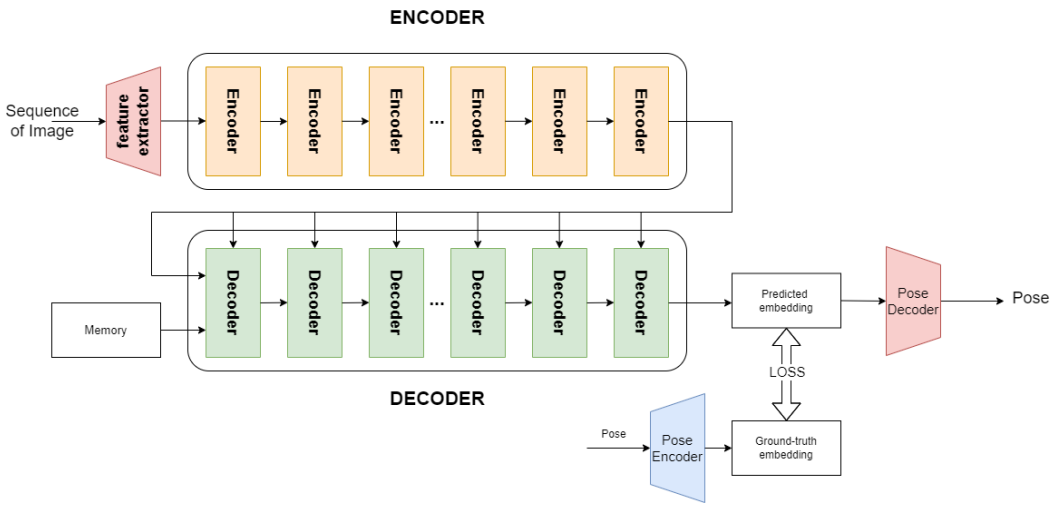

Figure 4.4: Encoder-Decoder with pose auto-encoder

4.1.4 Encoder-Decoder in autoregressive mode

In this model, we replaced the parameter vector *memory* with the output of the first part of the pose auto-encoder because the output is used to generate the ground-truth embedding used as target embedding, and the second part, as usual, is used to calculate the pose from embedding.

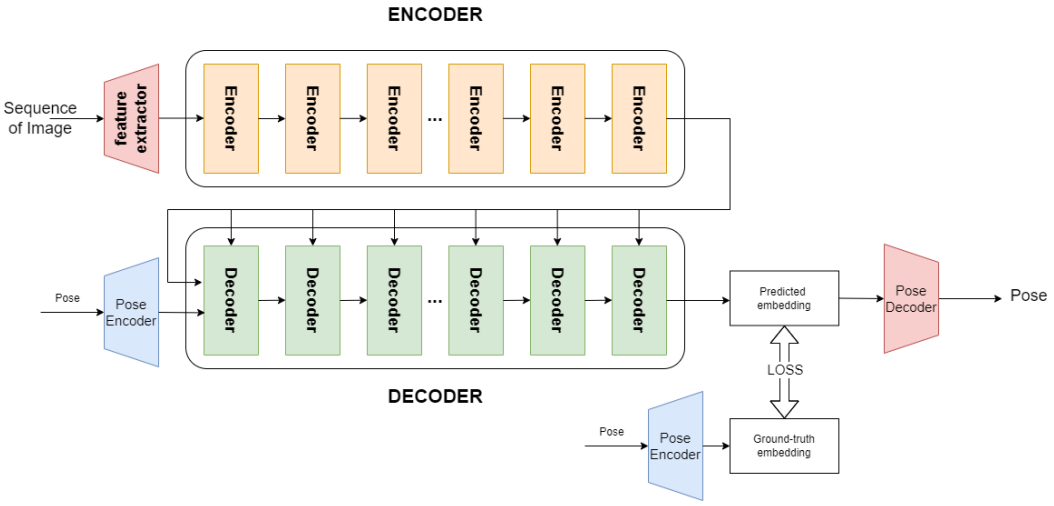

Figure 4.5: Encoder-Decoder with pose auto-encoder in autoregressive mode

In the figure 4.5, we can see the architecture of this model but the main difference between this model and previous ones is in the implementation of training and inference, this will be discussed in the next section.

4.2 Feeding Strategies

To solve the problem of visual odometry, we tried different approaches to feed the sequence of image into the model, and to construct the model itself. We tried following approaches to feed the data:

1. Feeding the sequence into the model directly and presenting the pose as *euler angles*.

2. Feeding the sequence into the model directly and presenting the pose as *rotation matrix* so with twelve numbers and *translation vector*.

3. Feeding the sequence into the model where the first frame is the origin of the reference frame and presenting the pose as *euler angles*.

4. Feeding the sequence into the model where the first frame is the origin of the reference frame and presenting the pose as *rotation matrix* and *translation vector*.

5. Feeding the sequence into the model where the first frame is the origin of the reference frame, and using the auto-regressive model to predict the pose.

We can divide these strategies into two groups: the first group is composed by strategy 1 and 2, and the second group is composed by strategy 3, 4 and 5. This division is because the first group uses the ground-truth without any preprocessing, meanwhile the second group uses the ground-truth translated with respect to the first pose of the sequence.

4.2.1 Directly feeding the sequence

With this strategy, we create each sequence by taking the images from the dataset and the pose as ground-truth without any processing. For the sake of simplicity, to explain the different strategy, we will use an example of trajectory which is as represented in the Figure 4.6:

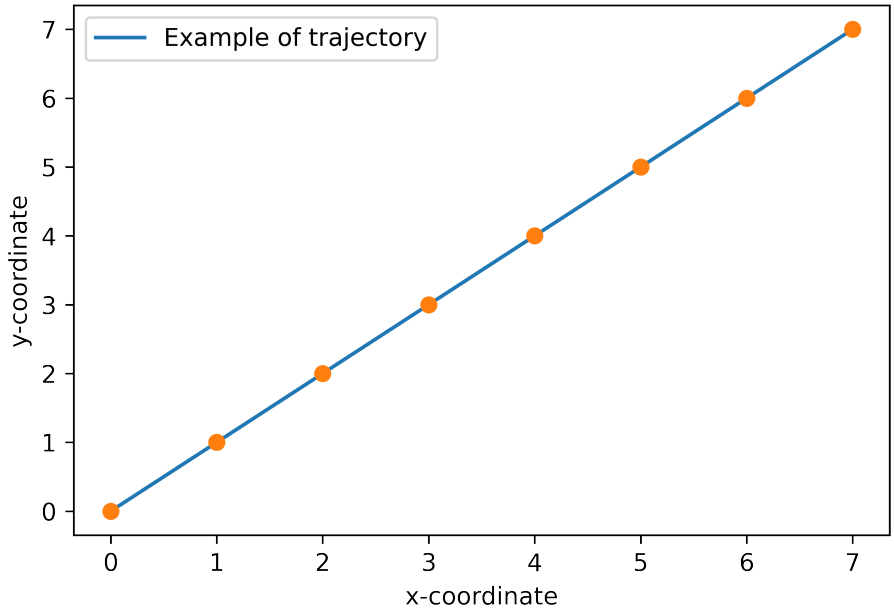

Figure 4.6: Example of the trajectory.

Of course, in the example the trajectory looks easy to predict because the distance between every pair of pose is the same, and trajectory direction is not changing. Meanwhile, in the KITTI dataset, the trajectories are more complex, and the distance between the poses is not constant, and the direction of the trajectory is changing. We can see the trajectory where each dot represents an image-pose pair. From this trajectory, we can represent the dataset and the creation of the sequences as follows:

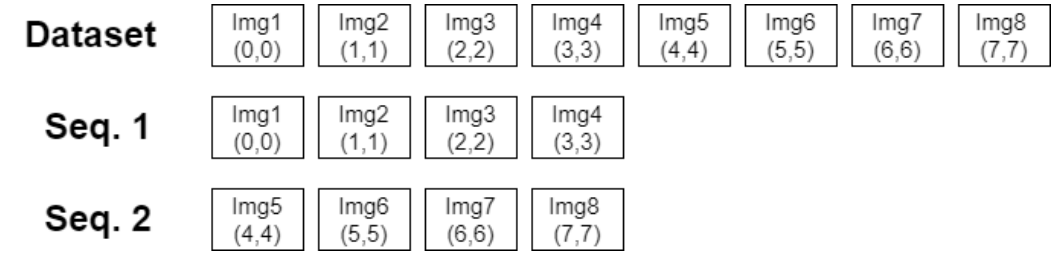

Figure 4.7: Example of the dataset and the sequences.

This way of feeding the data is the simplest one, but it has some drawbacks. The first one is that when we split the training dataset into different sequences, for each sequence, the coordinates do not start from the origin (i.e., (0,0)) but for example for the second sequence, it starts from (4,4). This is problematic for the model because it has to learn this kind of translation, and it is not a trivial task. The second drawback is that the model has to predict a very large interval of numbers in this case, the range goes from 0 to 7, but in real case, like in the KITTI dataset, the range goes from 0 to 1000. The third drawback is that, with a dataset of N images, we can have only $\frac{N}{seq\_len}$ sequences, where seq_len is the length of the sequence, this aggravates on the problem of small dataset available.

4.2.2 Sequence with origin

In this second strategy, we create each sequence by taking the images from the dataset, and the pose as ground-truth by performing a translation operation. In this way, the sequences can be represented as follows:

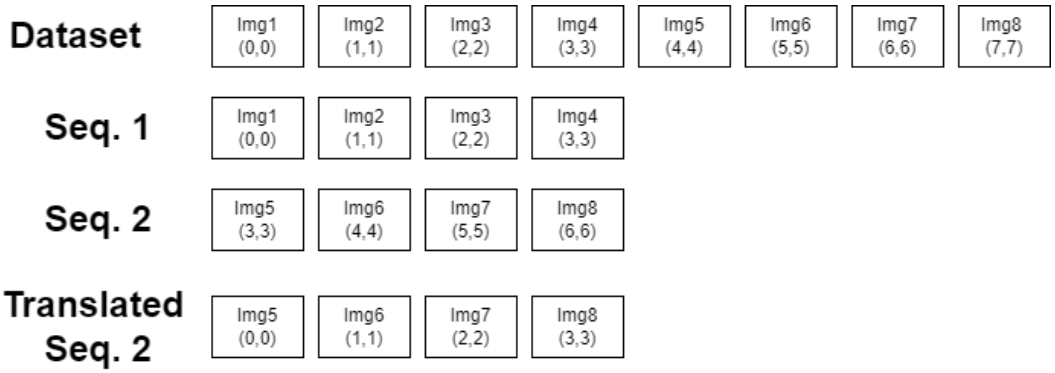

Figure 4.8: Example of the dataset with origin and the split of sequences.

We can see that the sequences start from the origin, and the model has to learn only the relative pose to the first frame of the sequence, instead of the whole trajectory. But this solution is not perfect, in the sense that, in this trajectory we

have eight image-pose pair, and with $seq_len = 4$ we can generate only two sequences, and considering that the Kitti dataset has about 20.000 images, this is a big problem.

To solve the previous mentioned problem, we devised a slightly modified strategy, which essentially takes every image-pose pair and creates a sequence of length seq_len, like represented in Figure 4.9:

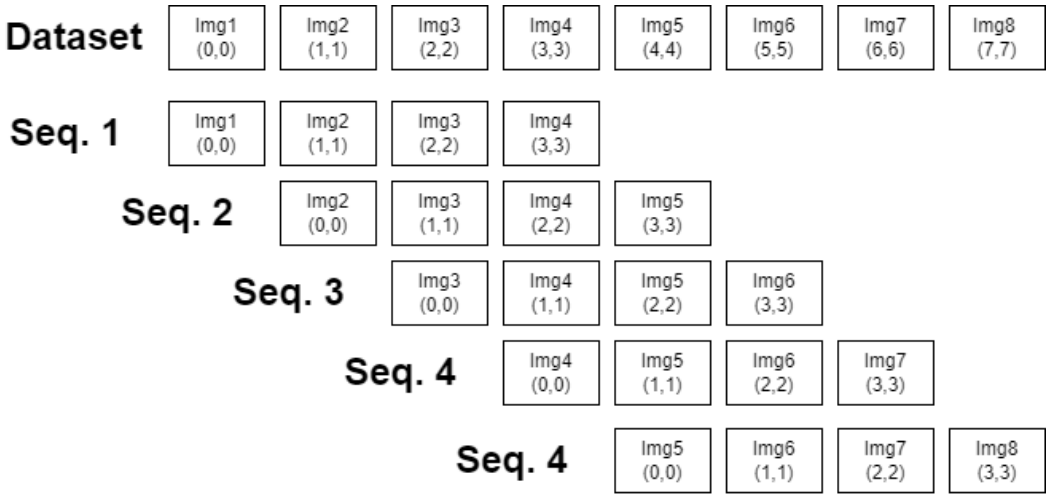

Figure 4.9: Example of the dataset with origin.

As represented in the previous figure, with a dataset of N images, we can have $N - seq\_len + 1$ sequences, and this is a big improvement with respect to the previous strategy. Also, in this way, we are doing a sort of data augmentation, because we're feeding the model with different sequences, and this is helpful for the model.

As comparison, with first strategy, using KITTI dataset, we can have only $\frac{N}{seq\_len}$ sequences, where $N = 20.000$ and $seq\_len = 4$, so we can have only 5000 sequences, while with the latest strategy, we can have $N - seq\_len + 1$ sequences, so we can have 19997 sequences, which is an important improvement.

Chapter 5

Implementations

In this chapter we will discuss about the implementations of different components of the project and the reason that led us to such choices.

5.1 Dataset preprocessing

Before feeding the sequence of images into the model, as it is usual in the literature, we need to preprocess the images.

The first operation is the normalization (dividing each pixel value by 255) in order to have the images in the range $[0, 1]$, this is useful because the neural network works better with this range as it has to compute smaller range of numbers. The second operation is the standardization, in this way, we are centering the data around zero, and we are reducing the variance of the data, to do so, we computed the mean for each of RGB channels following this formula:

$$\mu_c = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} I_c(i, j) \tag{5.1}$$

where $I_c(i, j)$ is the j-th pixel of the i-th row of the c-th channel, W is the width of the image, H is height of the image, μ_c is the mean of the c-th channel. Then, the standard deviation as follow:

$$\sigma_c = \sqrt{\frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} (I_c(i, j) - \mu_c)^2} \tag{5.2}$$

where σ_c is the standard deviation of the c-th channel. Finally, we normalized the images as follow:

$$I_c^*(i) = \frac{I_c(i, j) - \mu_c}{\sigma_c} \tag{5.3}$$

$\forall i \in [1, H], j \in [1, W], c \in [R, G, B]$ and $I_c^*(i)$ is the normalized i-th pixel of the c-th channel.

We perform the aforementioned operations to all images of the KITTI dataset saving new images, in this way, we need to process them just once, saving time during the training process.

5.2 Data Loading

The data-loading process is strictly connected to the §4.2 Feeding strategies, because it changes how we want to feed the data to the neural network.

This process is divided into two parts, the first one is the loading process, and the second one is how to draw a batch of sequences.

5.2.1 Loading process

Initially, we load the images and the poses into the RAM from the KITTI dataset, so in the same time we require the computer to keep in memory a lot of images. And, this is not efficient from a point of view of memory.

Then, we decided to store only the path to the images and the poses in the RAM, and we load the images and the poses from the disk when we need them. This is a good solution because we don't need to keep in memory all the images, but we need to load them from the disk when we need them. So the performance of the disk where we store the images is important.

To perform this, we created a class called *KittiDataset* that inherits from the *Dataset* class of the *PyTorch* library. And, by overriding the $\_\_getitem\_\_$ and $\_\_len\_\_$ method, we can load the images and the poses from the disk when we need them.

Then, the last operation we do is to resize the images to (1200, 360), this is to ensure that all the images has the same size.

5.2.2 Batch drawing

The batch drawing is the process of drawing a batch of items from the dataset. In our case, each item is a sequence of images and poses.

The API pytorch provides for this is the *DataLoader* class, which takes in input a dataset object. By default, the Dataloader class shuffles the dataset, and it draws a batch of items from the dataset, it processes them with *collate_fn* function, and it returns them.

The *collate_fn* function is a function that takes in input a list of items, and it returns a tuple of tensors. The first one is the input for the NN, so the images, and the second one is the target, so the poses.

Another important parameter of the class *DataLoader* is Sampler or BatchSampler. The difference between them is that with Sampler we can customize how to select items from the dataset, influencing the choice of the dataloader over items, meanwhile, the BatchSampler is used to customize how to select batches from the dataset, influencing the choice of the dataloader over one batch.

To implement the first strategy of §4.2.2 we implemented a BatchSampler that draws a batch of sequences, such that each item (sequence) is connected to the one before and after it.

But, for the final version (Second one introduced in §4.2.2), we use the default sampler, because we implemented the dataset $\_\_getitem\_\_$ method in a way that it returns a sequence of images and poses. E.g., when the $item_i$ is requested, the method returns a sequence from $item_i$ to $item_{i+seq_{len}-1}$. The $\_\_getitem\_\_$ method also processes the poses in a way the pose of the first image is the origin of the coordinate system and the origin of the sequence.

Code 5.1: The $\_\_getitem\_\_$ method of the *KittiDataset* class.

```
def __getitem__(self, idx):
    images, poses  = [], []
    to_translation, to_rotation = None, None
    for i in range(idx, idx + self.seq_len):
        img_path, pose = self.data[i]
        rot, tran = pose[:3, :3], pose[:, -1]
        if to_translation is None:
            to_translation, to_rotation = tran, rot
        tran = tran - to_translation
        rot = rot @ torch.t(to_rotation)
        pose = torch.cat((rot, tran.reshape(3, 1)), dim=1).
           reshape(-1, )
        frame = load_image(img_path, self.transform)
        images.append(frame)
        paths.append(img_path)
    return images, poses
```

In this way, we can randomly sample sequences from the dataset, removing the constraint of the connection between the sequences.

When we train the network, we set the shuffle parameter to **True**, to **False**

during th test time, because we want to have the sequences in the same order as in the dataset to be able to reconstruct the trajectory. To reconstruct the trajectory, we need to obtain the prediction of each sequence, then re-concatenate them as Code. 5.x shows:

Code 5.2: Reconstruct the trajectory from the predictions which is a list of batches of sequences.

```
def composing_trajectory(predictions):
    trajectory = []
    back_translation = predictions[0, 0, 0].reshape(3, 4)[:3,
       -1]
    back_rotation = predictions[0, 0, 0].reshape(3, 4)[:3, :3]
    trajectory.append(torch.cat((back_rotation.reshape(3, 3),
       back_translation.reshape(3, 1)), dim=1).flatten())
    for k in range(predictions.shape[0]):
        # number of batches
        for i in range(predictions.shape[1]):
            # number of sequence in each batch
            for j in range(predictions.shape[2]):
                # number of frames in each sequence
                current_pose = predictions[k, i, j].reshape(3,
                    4)
                rot, tran = current_pose[:3, :3], current_pose
                   [:3, -1]
                if j == 0: # beginning of the sequence
                    last_pose = trajectory[-1].reshape(3, 4)
                    back_translation = last_pose[:3, -1]
                    back_rotation = last_pose[3, :3]
                else:
                    tran = tran + back_translation
                    rot = rot @ back_rotation
                    trajectory.append(torch.cat((rot.reshape
                       (3, 3), tran.reshape(3, 1)), dim=1).
                       flatten())

    return torch.stack(trajectory)
```

Where the *predictions* is a tensor of shape *(num_batchs, batch_size, seq_len, 12)*, i.e., a list of batches.

5.3 Models

5.3.1 Standard model

We used PyTorch deep-learning framework to implement the models because it's flexible, intuitive, and easy to use. We mainly used ResNet18 and ResNet50 as **feature extractor**, we used the pre-trained version of these models, which are trained on ImageNet dataset by the authors of the library.

We took the model, removed the last layer, in this way, as output of the layer we obtain a tensor of size 512 or 2048, depending on the model, and we used this as the embedding.

Then, we used the *TransformerEncoderLayer*, *TransformerEncoder*, *TransformerDecoderLayer*, and *TransformerDecoder* from the PyTorch library. Where TransformerEncoder is single layer, TransformerEncoderLayer is the container for the encoder layers, and it takes as input the number of layers. Meanwhile, the TransformerEncoder needs as input:

- *embedding dimension*: 512 or 2048, depending on the feature extractor;
- *n_head*: the number of attention heads;
- *mlp_dim*: the number of neurons in the MLP;
- *drop_out*: the dropout rate;

The TransformerDecoder and TransformerDecoderLayer are similar to the TransformerEncoder and TransformerEncoderLayer, respectively. The difference between TransformerEncoderLayer and the TransformerDecoderLayer is that the latter in *forward* method, it takes as mandatory parameter the target sequence, while the former doesn't. There are also some optional parameter, like the *src_key_padding_mask* and *src_key_padding_mask*, which functionality is explained in §2.1.2.

Finally, depending on different version of the model we introduced in §4.1, we used different MLP to predict the pose.

5.3.2 Auto-regressive Model

Before starting with our implementation of the autoregressive model, we want to explain the intuition behind it. An autoregressive (AR) model is a model that predicts future behavior based on past behavior. It's used for forecasting when there is some correlation between values in a time series and the values that precede and succeed them. In an AR model, the value of the outcome variable (Y) at some point

t is directly related to the predictor variable (X). The AR process is an example of a stochastic process, where you can have a degree of uncertainty or randomness built into the model. The randomness means that you might be able to predict future values of the outcome variable, but you can't be certain of the exact value. AR models are also called conditional mean models, Markov Models, or transition models.

In our case, what we want is to predict the future poses of the sequence, given the past poses. In details, we want to predict the pose at time t_i, given the poses at time $t_0, t_1, \ldots, t-i-1$. To do this, we feed the network differently depending on the training phase and the inference phase. In the first one, we feed the network with the output of the transformer decoder with the output of the encoder, together with the target which is constructed by the following function:

Code 5.3: Training target

```
def _training_target(self, gt):
    """
    :return: the target for the training phase, that should be
        :
    [
    [GT, trg, trg, trg, trg],
    [GT, GT,  trg, trg, trg],
    [Gt, GT,  GT,  trg,  trg],
    [Gt, GT,  GT,  GT,  trg]
    ]
    """
    learnable_target = repeat(self.learnable_target, 'n d -> b
        n d', b=gt.shape[0])
    lower_triangular = torch.tril(gt)
    upper_triangular = torch.triu(learnable_target, 1)
    return lower_triangular + upper_triangular
```

In the schema, each row is a sequence, and each element of the row is a pose. GT is the ground truth pose embedding computed by the pose auto-encoder, meanwhile trg is the memory of the transformer decoder. In essence, the batch_size is 4 and seq_len is 5. With this target, at the time 0, we have the GT as first pose, also because it's considered the origin of that sequence, then what we have to predict is the second pose. At the time 2, we have the GT at pose 0 and 1, and we have to predict the pose at time 2. And so on so forth. Last thing we should pay attention to is that the loss should be computed only on the elements on the first diagonal of

the matrix, because in the lower triangular matrix, we have the ground truth, and we don't care about the predictions of the upper triangular matrix.

For the inference phase, we feed the network with the output of the transformer decoder with the output of the encoder, together with the target which is constructed by the following function:

Code 5.4: Inference target

```
def _inference_target(self, predictions, column_index):
    """
    column_index is the index of the column that we want to
       predict: always 0 < x < seq_len
    :return: the target for the inference phase, that should
       be:
    for column_index = 1:
    [
    [GT, trg, trg, trg, trg],
    ...
    [GT, trg, trg, trg, trg],
    ]
    for column_index = 2:
    [
    [GT, GT, trg, trg, trg],
    ...
    [GT, GT, trg, trg, trg],
    ]
    """
    # origin_embed shape: batch_size, 1, emb_size,
    # learnable_target shape: batch_size, seq_len, emb_size
    # predictions shape: batch_size, seq_len, emb_size
    return torch.cat((predictions[:column_index, :], self.
       learnable_target[column_index:, :]), dim=0)
```

The the code for inference is the following the one shown in **Code xx**.

Code 5.5: Inference

```
predictions = torch.zeros_like(x)
predictions[:, 0, :] = gt_target  # embedding of the origin
for i in range(1, x.shape[0]):
    new_target = self._inference_target(predictions[i], i).
       unsqueeze(0)
    predicted = self.decoder(new_target, x[i].unsqueeze(0),
```

```
        tgt_mask=generate_upper_triangular_mask(x.shape).to(
        self.device))
    predictions[i, :, :] = predicted[:, :, :]
    if i < x.shape[0] - 1:
        predictions[i + 1, :, :] = predicted[:, :, :]
```

In the previous code snippet, the *gt_ target* origin embedding, we instantiate a tensor with the same shape of the input, we set the first column to the origin embedding, and then we iterate over the sequence. At each iteration, we predict the pose at i-th, given the poses at time $t_0, t_1, \ldots, t - i - 1$ for all the sequences of the batch. Differently from the training, we should compute the loss for the whole *predictions* tensor, because we are predicting all the poses.

5.4 Losses

Loss function is an important component of the training process, it is used to evaluate the performance of the model, and by optimizing it (minimizing it or maximizing it) we can improve the performance of the model. In our case, we tried different loss functions:

- Mean Squared Error (MSE);
- Weighted MSE;
- Loss derived from Absolute Trajectory Error (ATE) and Relative Trajectory Error (RTE);

5.4.1 Mean Square Error (MSE)

MSE is the most common loss function used in regression problems. Given the ground truth pose p_{gt} and the predicted pose p_{pred}, the MSE is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(p_{gt} - p_{pred})^2 \tag{5.4}$$

This loss function is derived from the square of Euclidean distance, it is always positive value that decreases as the error approaches zero.

5.4.2 Weighted Mean Square Error-WMSE

WMSE is a variation of the MSE, which is useful when we want to give more importance to some components of the pose. For example, given a sequence of poses:

p_0, p_1 ... p_n, we want to give more importance to the poses more distant from the origin, because we want the poses far away from the origin to be closer to the ground truth. In this way, we impose the network to give more importance for the poses far away from the origin as the errors accumulate over steps. The formula of the WMSE is:

$$WMSE = \frac{1}{N}\sum_{i=1}^{N} i * (p_{gt} - p_{pred})^2 \tag{5.5}$$

In the equation, we chose i as the weight of the pose, but we can use any other function of i.

5.4.3 Loss derived from ATE and RTE

todo write the section

5.5 Pose Auto-encoder

Pose encoder structure is very simple, it takes as input the pose, and it outputs the same pose. We devised this a MLP can be get in input the pose, and it can reproduce the same poses with small margin of error. In fact, we discovered that with §4.2.1, the auto-encoder really struggles to reduce the loss, MSE in this case, the reason is that the network is required to output a very large range of values. To fix this, we are forced to used §4.2.2. Another reason is that by use ReLU activation function, the network is forced to output only positive values, meanwhile for some poses, the output of the network is negative. To solve this problem, we used LeakyReLU activation function, which allows the network to output negative values.

We tried with different shapes for the auto-encoder, but we found that the best results are obtained with this structure: [x, 64, 128, 256, 512, 512, 512, 512, 512, 512, 512, 512, 256, 128, 64, x] where x is the dimension of the input, we used $x \in \{6, 12\}$. The structure of the auto-encoder is obtained growing the number of layers, at the beginning we used only 2 layers with 512 neurons, but in that case the capacity was not enough, causing the loss to be very high. Then by increasing the number of layers, we found that the best results are obtained with 16 layers.

The network was trained with MSE loss, ADAM optimizer and learning rate scheduling, as training set we used the KITTI dataset sequences, but the number 3, which is used as validation set. On training set, we achieved the loss of 0.0001, while on validation set, we achieved the loss of 0.0002.

So, given the low loss-value on the testing set, we can conclude that the auto-encoder is able to reproduce the same poses with a small margin of error. Therefore, we can use it to produce the embedding for the poses, and used it as the ground truth embedding.

5.6 Training cycle

In this section, we will present the training cycle used to train the network. The whole project is divided into three phases:

1. Training;
2. Validation;
3. Testing.

In the *main.py* file there is a class called *Main* that contains three main methods, one for each phase.

In the *__init__* method, we initialize the network, the optimizer, the loss function, the dataset, the dataloader, log-writer, schedulers, and other smaller components. Each time we start a training, we create a new folder for that run, and we save a copy the configuration file which contains all the parameters used in that run.

5.6.1 Training and Validation

At the beginning we print all the parameters used in that run, and we start the training cycle. We start by loading the dataset and the dataloader, then, the training cycle starts. Which iterate for a certain number of epochs, and for each epoch, we iterate over the dataset.

For each epoch, the network is trained on the whole dataset, and the training loss is computed. then, we validate the network on the validation dataset, and we compute the validation loss. At the end of each epoch, we save the network weights, the optimizer state, the scheduler state, and we log the learning rate, the training loss, validation loss, and we plot the trajectory predicted during the validation.

5.6.2 Testing

For the testing, the process is pretty much the same, with only difference that the testing dataloader does **NOT** shuffle the data. It plots the trajectory predicted

by the network, and it computes the loss on the testing dataset. Then, it evaluates the network prediction by computing the **RPE** and **ATE** metrics.

Chapter 6

Final discussions

In this chapter we will discuss the results achieved, future developments and personal comments.

6.1 Results

6.1.1 Full sequence prediction

With different models a variety of results are achieved, but the most important result is that an important baseline for the future development of this kind of systems has been settled down. After a few trials with few models, which produced some circular trajectories, with the encoder-only version of transformer, and feeding the *sequence 3* of Kitti (for more details § 3.1) where the first image is considered as origin, we showed that the model is able to learn a single sequence in over-fitting, but fails when trying to over-fit a more complex sequence.

The encoder-decoder version achieved the same results as the previous version, but the model was able to learn also the *sequence 7* of Kitti, but it fails when trained with both *sequence 3* and *sequence 7*.

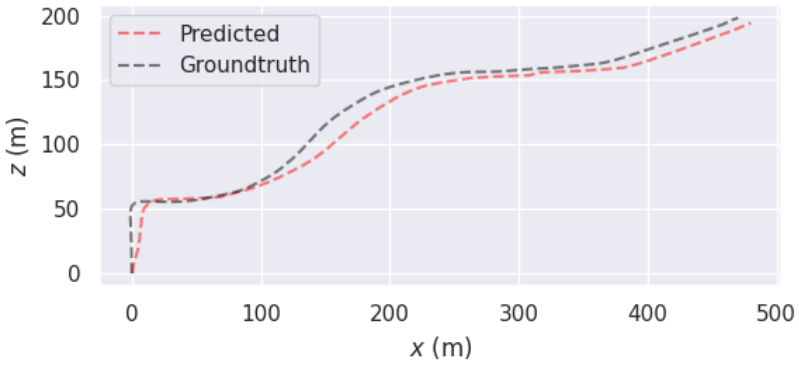

Figure 6.1: Good prediction sequence 3

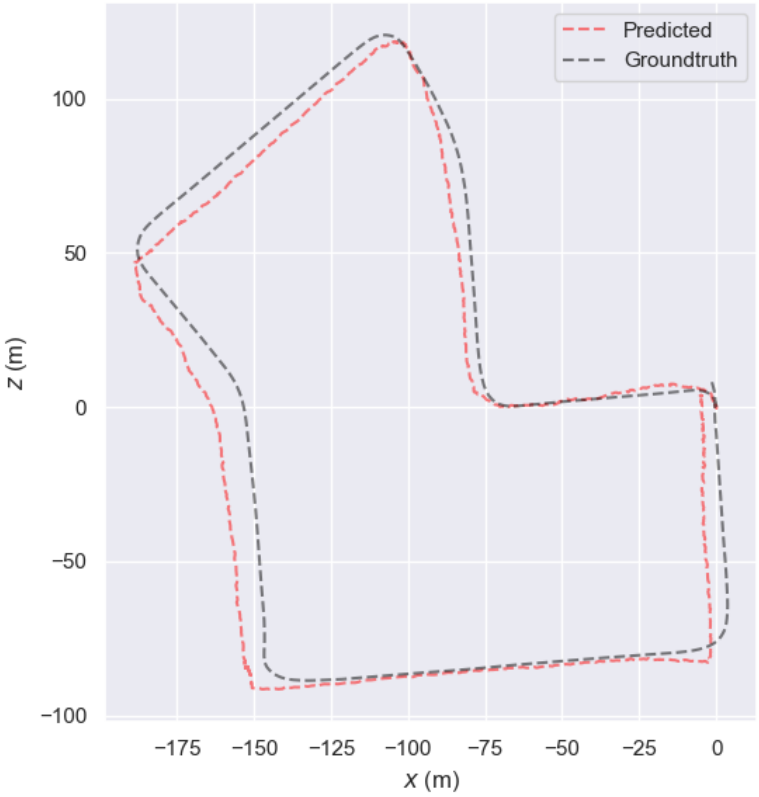

Figure 6.2: Good prediction sequence 7

6.1.2 Autoregressive models

We implemented only the encoder-decoder version of the transformer in the autoregressive way, and most of the time the prediction of the network during the training on seq 3 is just a straight line. So the model is **not** able to predict the simplest sequence in over-fitting. The model could not predict any reasonable trajectory, predicting only a linear trajectory as the follow:

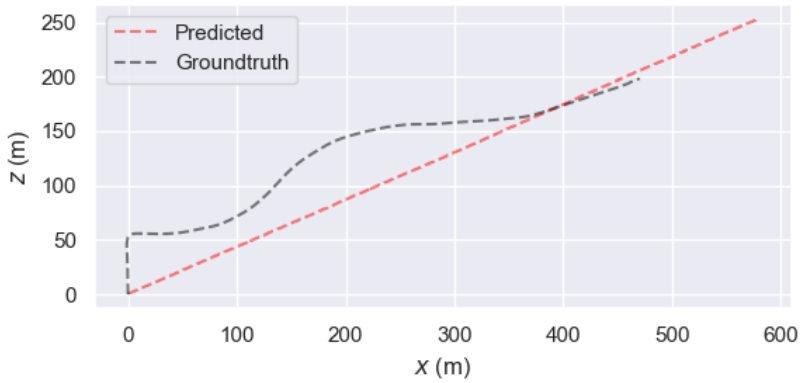

Figure 6.3: Bad prediction sequence 3 of autoregressive model

Although the model has been trained for more than two hundred epochs, the network cannot understand the goal, and this maybe is due to the loss function.

6.2 Knowledge Acquired

Transformer Logging Methodology

6.3 Future Developments

Change the input strategy, using a pair of images Change the loss function Change the architecture of the network change the feature extractor change the prediction head, consider using RNN

6.4 Personal Evaluation

Glossary

ATE ATE is 51

CNN Convolutional Neural Networks are a class of neural network which uses the convolution. The convolution is a LSI operator, which given an input array and a kernel, it performs the summation of the element-wise product between elements of the kernel and those of the input, then it slides the kernel over the whole range of the input. . 51

IMU Inertial measurement unit is a device that measures the motion of an object in space.. 51

MHA Multi head attention is a module for attention mechanisms which run through an attention mechanisms several times in parallel. The independent attention outputs are then concatenated and linearly transformed into the expected dimension.. 51

NLP Natural Language Processing is 51

NN Neural network models are a subset of Machine Learning models. NN is a network based on the concept of artificial neuron and neurons are organized in layers. The aim of the network is to mimic the data that is used in the training time. . 51

ReLU ReLU is 51

RTE RTE is 51

SLAM Slam is a computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it.. 51

Vanishing gradient problem When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss functions approaches a value close to zero, and the partial derivative vanishes.. 9, 49

Word Example of a term in the glossary. 6, 49

Acronyms

ATE Absolute Trajectory Error. 40

CNN Convolutional Neural Network. 9

IMU Inertial measurement unit. 4

MHA Multi-Head attention. 9

NLP Natural Language Processing. 9

NN Neural network. 4

ReLU Rectified Linear Unit. 11

RTE Relative Trajectory Error. 40

SLAM Simultaneous Localization and Mapping. 7

Bibliopraphy

Paper references

Alkendi, Yusra, Lakmal Seneviratne, and Yahya Zweiri (May 2021). "State of the Art in Vision-Based Localization Techniques for Autonomous Navigation Systems". In: *IEEE Access* PP, pp. 1–1. DOI: 10.1109/ACCESS.2021.3082778 (cit. on p. 14).

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: DOI: 10.48550/ARXIV.1512.03385. URL: https://arxiv.org/abs/1512.03385 (cit. on p. 8).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: 25. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on p. 7).

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: URL: https://arxiv.org/abs/1409.1556 (cit. on p. 7).

Szegedy, Christian, Wei Liu, et al. (2014). "Going Deeper with Convolutions". In: URL: https://arxiv.org/abs/1409.4842 (cit. on p. 7).

Szegedy, Christian, Vincent Vanhoucke, et al. (2015). "Rethinking the Inception Architecture for Computer Vision". In: URL: https://arxiv.org/abs/1512.00567 (cit. on p. 8).