POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Towards Real-Time Inference: A Fusion of Pose Estimation and Object Tracking

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Greta Corti**

Student ID: 970485
Advisor: Prof. Matteo Matteucci
Co-advisors: PhD. Francesco Lattari
PhD. Simone Mentasti
PhD. Riccardo Santambrogio
Academic Year: 2022-23

Abstract

In the rapidly evolving field of computer vision, accurate object pose estimation models have exhibited exceptional prowess in predicting spatial orientations. However, seamlessly integrating these models into real-time scenarios, particularly on resource-constrained wearable devices, poses a formidable challenge due to their computational demands. This work addresses this challenge through a multi-faceted approach, offering a novel framework that unifies pose estimation and object tracking to enhance inference speed to real-time. Noteworthy contributions include a comprehensive quantitative analysis of diverse embedded platforms, the introduction of this innovative framework, and the development of a flexible, lightweight deep-learning-based network for object tracking. The framework strategically leverages both pose estimation and object tracking, with the latter exemplified by SwiftTrack, an innovative model ensuring high-speed and precise pose estimation for wearable devices.

Keywords: 6D pose estimation, object tracking, real-time inference, embedded platforms

Abstract in lingua italiana

Nel dinamico campo della computer vision, i modelli accurati di stima della posa degli oggetti hanno dimostrato eccezionali capacità nel predire orientamenti spaziali. Tuttavia, l'integrazione fluida di tali modelli in scenari in tempo reale, specialmente su dispositivi indossabili con risorse limitate, rappresenta una sfida considerevole a causa delle loro esigenze computazionali. Questa tesi si propone di affrontare questa sfida attraverso un approccio innovativo, presentando un nuovo framework che unisce la stima della posa e il tracciamento degli oggetti per migliorare la velocità di inferenza in tempo reale. Tra i contributi significativi, troviamo un'analisi quantitativa completa di diverse piattaforme embedded, l'introduzione di questo innovativo framework, e lo sviluppo di una rete deep-learning leggera e flessibile per il tracciamento degli oggetti. Il framework sfrutta strategicamente sia la stima della posa che il tracciamento degli oggetti, con quest'ultimo esemplificato da SwiftTrack, un modello innovativo che garantisce una stima della posa ad alta velocità e precisa per i dispositivi indossabili.

Parole chiave: stima della posa 6D, tracciamento degli oggetti, inferenza in tempo reale, piattaforme embedded

Contents

Introduction

In this chapter, we provide a comprehensive introduction to the key topics explored in this thesis. By highlighting the challenges we faced and the subsequent solutions proposed, we aim to address a pertinent problem in the field. This research contributes substantially to existing knowledge, and we will outline the main achievements. Finally, we offer an overview of the entire paper structure, guiding readers through the organized presentation of our findings.

0.1. Problem statement

In recent advancement in computer vision, object pose estimation models have demonstrated exceptional accuracy in capturing and predicting the spatial orientations of various objects. Despite these improvements, the practicality of implementing such highly accurate models in real-time scenarios, particularly on wearable devices, presents an impressive challenge.

The issue lies in the complex computational processes that are integral to these sophisticated models. Models designed to recognize detailed spatial information often come with a high parameter count, resulting in computationally intensive processes. In the context of real-time applications, especially those demanding rapid decision-making or interaction, this computational demand becomes a critical bottleneck. The delay introduced by these intricate models hampers their ability to provide timely results, a crucial factor in applications such as robotics, augmented reality, or automated manufacturing. Furthermore, deploying accurate pose estimation models in resource-constrained environments, like edge devices or embedded systems found in wearables, proves challenging. These wearable devices typically lack the computational power required for real-time pose estimation, creating a gap between accurate models in controlled settings and their practical application in dynamic, real-world scenarios.

This challenge is particularly pronounced in wearable applications, such as augmented reality glasses, where rapid and precise spatial information is paramount for user experience and functionality.

0.2. Contribution

The main contributions of this work are as follows:

- A quantitative analysis and comparison of different GPU hardwares available on the market.
- A novel definition of a framework that unify a pose estimation method with an object tracking algorithm to increase the speed of inference to real time.
- A flexible and lightweight deep-learning based network for object tracking that is capable of predicting bounding box and pose exploiting RGB images and performs at high time inference.

0.3. Document structure

The thesis is organized as follow:

1. **State of the Art**: In Chapter 1, we explore existing deep neural network models for 6D object pose estimation and object tracking. We describe their main structure and their characteristics. Then, we present the different pose representations utilized in computer vision and provide information about the geometry of a camera. Lastly, we address the challenges and identify key datasets that drive progress in the fields.
2. **Proposed Method**: Chapter 2 introduces our new deep learning-based framework, which serves as the solution to our problem statement. We describe in detail the architecture, input, and output of the object tracking model within the general framework.
3. **Preliminary Analysis on Embedded Platforms Timing**: Chapter 3 presents an analysis of two deep-learning neural networks for pose estimation on embedded platforms. We begin by comparing their efficiency and on the same dataset and then asses their performance on CPU, GPU, and NVIDIA Jetson boards.
4. **Experiments and Results**: Chapter 4 details all the experiments conducted to make our object tracking model efficient and fast. We begin by discussing the dataset utilized and its subdivision on training and validation datasets. We then explore the metrics used to evaluate the experiments and provide technical details about the implementation. In conclusion, we delve into variations in loss function, data augmentation, multi-task learning, backbone comparison and offer final remarks on the proposed framework.

5. **Conclusions and Future Developments**: Chapter 5 summarizes the key findings with reflections on the entire study and proposes potential areas for future research and development.

1 | State of the Art

In this chapter we present an overview of state-of-the-art methods for 6DoF object pose estimation and tracking. In the next sections, we describe how 6D object pose estimation and tracking algorithms work, the main classifications in which they are distinguished, and some examples of existing implemented frameworks. After that, we define and analyze the possible representations of the 6 DoF pose. Then we investigate the main challenges that can be encountered when dealing with these types of algorithms. Finally, we get an insight into some of the datasets used in this area.

1.1. 6D Object Pose Estimation

6D object pose estimation refers to the task of determining the precise position and orientation (*pose*) of a three-dimensional object in a six-dimensional space, as represented in Figure 1.1. The six dimensions represent the object's translation along the X, Y, and Z axes, as well as its rotation around these axes.

The goal of 6D object pose estimation is to accurately estimate the position and orientation of an object in the real world based on the information provided by sensors such as cameras or depth sensors. This task is essential in various fields, including robotics, augmented reality, virtual reality, and autonomous navigation.

Typically, 6D object pose estimation involves several steps:

1. **Object detection**: detecting and localizing the object of interest within the sensor's field of view.
2. **Feature extraction**: extracting relevant features from the object, such as keypoints or descriptors, to represent its appearance and geometry.
3. **Pose estimation**: estimating the object's pose by matching the extracted features to a known model or template of the object.
4. **Refinement**: iteratively refining the pose estimation using optimization techniques to minimize the difference between the observed features and the model's

Figure 1.1: 6 DoF pose estimation with 3D bounding box representation (from [26])

predictions.

Various algorithms and techniques can be used for 6D object pose estimation, including geometric methods [6], template matching [35], machine learning-based approaches [42], and deep learning methods [37]. These techniques leverage the information from sensors to solve the estimation problem, often relying on computer vision algorithms and mathematical models. In this thesis, we focus especially on deep learning methods by depending on computer vision methods to obtain meaningful information from digital RGB images without utilizing depth information.

1.1.1. An Extensive Review of Techniques Based on Input Modalities

6 Degrees of Freedom (DoF) object pose estimation can be achieved using various techniques, traditional approaches [35] or deep learning-based methods [26]. The literature distinguishes and classifies the methods employed based on the input received by the network:

RGB-based methods: First methods rely on detecting and predicting 2D keypoints and then use the PnP algorithm [9] to calculate the poses. Nowadays, thanks to the advent of deep learning and the rise of Convolutional Neural Networks (*CNNs*), monocular 6D pose estimation achieved promising results. The main challenge of these networks is the lack of depth information: they have to estimate the 3D pose of an object from a single 2D image. This is an ill-posed problem because many possible 3D poses could correspond to a given 2D image. The lack of depth information makes it difficult for the network to disambiguate between these possibilities. Exists different kinds of approaches to overcome this problem. GDR-Net [37] introduces a network architecture that combines geometric guidance - the geometric relationship between object keypoints and their 3D model coordinates obtained by CAD models - and direct regression of the 6D pose parameters from the input image; while the network described in [26] leverages 3D bounding box annotations without requiring depth information. Other works such as [6] instead propose a method that exploits object aspect classification: they categorize objects based on shape and appearance, providing additional information for pose estimation.

Point Clouds-based methods: Point clouds are dense collections of 3D points in space that represent the shape and geometry of objects. With the reduction in cost and improvement in the accuracy of depth sensors, numerous techniques based on point clouds have been introduced. Typically, popular approaches employ 3D ConvNets [34] or 3D point cloud network [31] to extract the feature and estimate the 6D pose. However, these methods have limitations due to exorbitantly expensive computation, and inherent noise and deficiencies in point cloud data. They struggle to achieve satisfactory results with sparse and poorly textured point clouds, highlighting the importance of incorporating RGB data.

RGB-D-based methods: These neural networks are trained on images that contain both color and depth information. Classical methods extract 3D features from the input RGB-D data, group together points that belong to the same object (*correspondence grouping*), and verify if these groups of points correspond to an object in the scene or not (*hypothesis verification*) [1]. Instead of using hand-coded features, most data-driven methods now use deep neural networks to extract features due to the development of deep learning and their ability to deal with unstructured data. Some methods such as [36] fuse 3D data to 2D appearance features preserving the geometric structure of the input space. They use 2D CNNs for RGB images and point cloud networks for depth data considering the data diversity between them. Then, they fuse these features into one to predict poses. Unlike the previous approach, [18] proposes a method named Uni6D which takes both UV data and RGB-D images as explicit inputs. This way they do not need two different

networks to extract the RGB and depth information, but they simply need a single CNN with two extra heads for directly predicting 6D pose.

1.1.2. A Comprehensive Exploration of Two 6D Object Pose Estimation Algorithms: GDR-Net and DenseFusion

In this section, we provide a detailed examination of GDR-Net, an RGB-based network, and DenseFusion, a strong RGB-D-based model, which are two important algorithms utilized in the field of 6D object pose estimation. Through our analysis, we elucidate each algorithm's intricate methodologies, structural nuances, and performance characteristics. Chapter 3 expands on this exploration by thoroughly assessing the accuracy and inference speed of these algorithms on various embedded platforms. Our discussion of GDR-Net and DenseFusion delves into their distinct methods for managing input data, differences in architecture, and the effects of depth information on pose estimation precision. We specifically chose to evaluate these two algorithms to investigate whether the variation in input data provided to the network has a significant impact on enhancing efficiency and speed during inference.

Geometry-guided Direct Regression Network (GDR-Net)

The model presented in [37] learns the 6D pose in an end-to-end manner from dense-correspondence-based intermediate geometric representations. Given an RGB image and a set of N objects along with their corresponding 3D CAD model, the goal is to estimate the 6D pose of each object with respect to the camera (Figure 1.2). First, all objects in the scene are surveyed and zoomed in corresponding Region of Interest (RoI) which is given as input to the proposed network to infer several intermediate geometric feature maps. From there, the network derives the 6D pose of the object. For each zoom-in, the model infers three intermediate geometric feature maps with size 64x64: the first one is called *Surface Region Attention (M_{SRA})*, the second one is *Dense Correspondence Map (M_{2D-3D})* and the last one is named *Visible Object Mask (M_{vis})*. The Dense Correspondences Map M_{2D-3D} encodes the 2D-3D correspondence and information about the geometric shape of the object while the Surface Region Attention Map M_{SRA} represents the symmetry of an object, thus using M_{SRA} not only mitigates the influence of ambiguities but also facilitates M_{3D} learning by first locating coarse regions and then regressing finer coordinates. Using M_{2D-3D} and M_{SRA} the network obtains the 6D pose of the object thanks to the Patch-PnP 2D convolutional module: it consists of three 3x3 kernel convolutional layers followed by a Group Normalization [40] and ReLU activation.

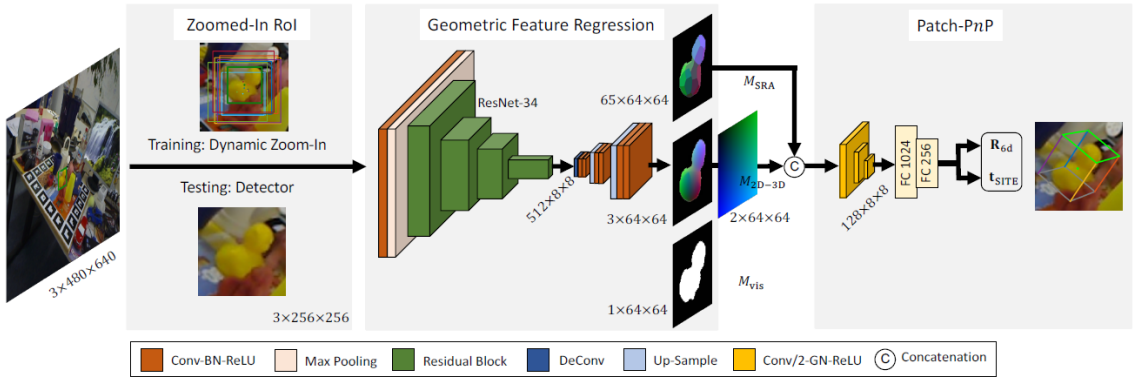


Figure 1.2: Framework of GDR-Net (from [37])

Two Fully Connected layers are applied to the flatten layer to reduce the size to 256. In the end, two parallel fully connected layers output rotation and translation of the object.

DenseFusion

Wang et al. [36] describes a generic framework for estimating the 6d pose of a set of known objects present in RGB-D images with cluttered scenes. Given the different nature of the color and depth information in the image, they are processed in two separate networks. The architecture (Figure 1.3) contains two stages: in the first one the network performs a semantic segmentation (*encoder-decoder architecture*) for each known object category present in the input color image, then for each object, the model uses the cropped image from the bounding box and the masked point cloud for the second step. The second step goes to estimate the 6d pose of the object. It consists of 4 components:

- A fully convolutional network that takes as input the cropped image of the object and extracts features on the color per pixel so that 3D point features and image features can be matched.
- A PointNet-based network [30] that processes each point in the point cloud to obtain geometric features of the object.
- A pixel-wise fusion network that combines the dense features obtained from the RGB image and the point cloud, and estimates the 6D pose of the object based on an unsupervised confidence score.
- An iterative self-refinement methodology to train the network in a way that learns and refines the estimation result iteratively.

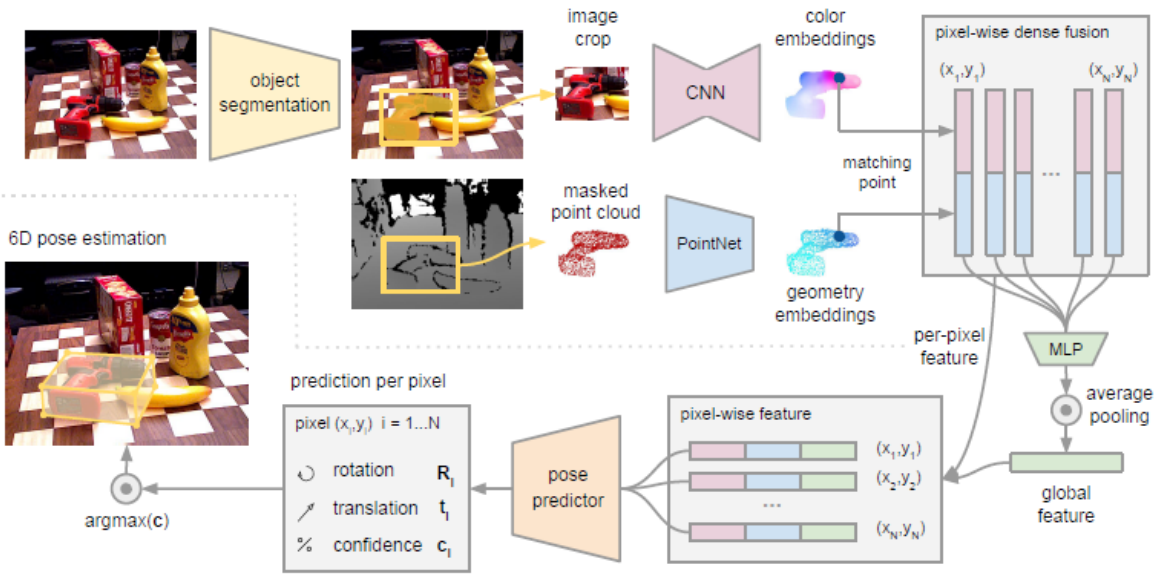


Figure 1.3: Overview of DenseFusion model (from [36])

The main difference between these two approaches is the type of data input: the first one needs only an RGB image, while the second one also needs depth information. Another difference is the presence of a refinement stage on the DenseFusion network to have a more accurate pose estimation of the object. According to the authors of the papers, both algorithms can be used for real-time object pose estimation and are robust in estimating the pose of objects in occluded ambient. In the following chapter, we will analyze and discuss the distinctions between these two approaches, specifically focusing on the speed of estimation with varying hardware.

1.2. Object Tracking

Object tracking is the procedure of precisely determining the position and monitoring the movement of a particular object or multiple objects in a sequence of images or frames captured by a camera or sensor. It involves identifying and monitoring the movement of the object of interest over time as it changes position, size, and orientation.

Like 6D object pose estimation, object tracking is frequently used in several fields such as robotics, surveillance, autonomous vehicles, and augmented reality.

The process of object tracking usually involves the following steps:

1. **Object initialization**: once the object is detected in the first frame or image of the sequence, its location and features are extracted to create an initial target

representation. This representation could include characteristics like color, texture, shape, or a combination of these.

2. **Target localization**: in subsequent frames, the task is to locate the object by searching within a specific region of interest (RoI) or the entire frame.
3. **Target re-identification**: object tracking also involves maintaining the identity of the object across frames, especially in scenarios where multiple objects are present.
4. **Motion prediction and filtering**: to handle occlusions, scale changes, or abrupt object movements, motion prediction and filtering techniques are employed. These methods estimate the future position of the object based on its previous motion patterns, enabling smoother tracking results.
5. **Track maintenance**: object tracking systems often incorporate mechanisms to handle situations where an object is temporarily lost due to occlusions or other challenges.

Object tracking algorithms can vary depending on the specific requirements of the application, the type of objects being tracked, and the available sensor data. Machine learning techniques, such as deep learning and online learning, have significantly advanced the field of object tracking in recent years, enabling more precise and robust tracking in complex scenarios.

1.2.1. Types of Object Tracking

Object tracking encompasses various methodologies and techniques that can be employed to achieve accurate and reliable tracking of objects. The first way to distinguish object tracking algorithms is to consider the nature of the data being processed:

- **Image tracking**: these algorithms analyze a single image at a time and determine the presence and position of the object(s) of interest within that image. The image's trajectory within the surrounding space is persistently monitored and traced.
- **Video tracking**: these models investigate the temporal dimension by tracking the object(s) across frames, estimating their motion, and maintaining their identity and position over time.

Both image and video object tracking methods can further be classified based on the number of objects they are tracking:

- **Single object tracking (SOT)** [14]: it aims to track only one target at a time during a sequence of images or a video. The target bounding box is defined in the

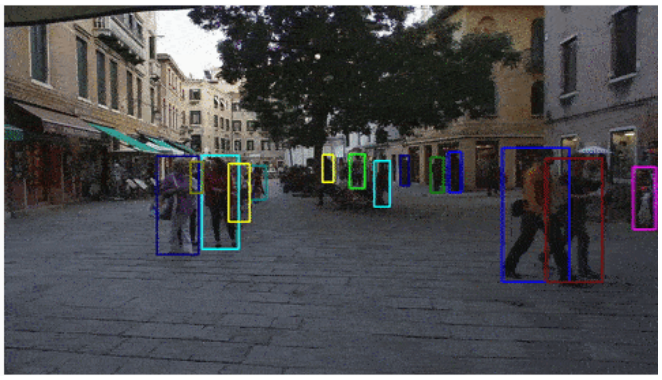
first image or frame of a video, and it is tracked in the rest of the images or frames. SOT algorithms should be able to track any detected object that is given, even an object on which no available classification model was trained.

- **Multiple object tracking (MOT)** [43]: it aims to track every single object of interest in a video. First of all, MOT algorithms must determine the number of objects in each frame; keep track of each object's identity from one frame to the next one until they leave the frame.

1.2.2. Deep Learning-based Approaches

Object tracking has been around for nearly 20 years, and numerous methods and ideas have been developed to increase the accuracy and efficiency of tracking algorithms. Some of these methods [12, 15, 17] used classical machine learning approaches such as k-Nearest Neighbour [21] or Support Vector Machine (SVM) [4]. These strategies are effective for predicting the target object, but they require the extraction of important and distinctive features by professionals.

(a) A single object (i.e. the car) being tracked. [32]



(b) Multiple people tracked. [10]

Figure 1.4: Examples of single object and multiple objects tracking.

On the other hand, deep learning models extract these features and representations on their own increasing the accuracy of object tracking in the scene.

In recent years, researchers extensively studied the utilization of deep learning for improving accuracy in object tracking. Several methodologies have been suggested by researchers, including [11, 14], that focus primarily on the detection of an object's position in an image. Nevertheless, in specific circumstances, it is necessary to track both the position and pose of the object. Researchers have proposed object pose tracking methods [23, 44] that find application in practical areas such as robotic grasping, autonomous driving, and augmented reality. Accordingly, object pose tracking has emerged as a vital aspect of object tracking. The precision and speed of techniques used for tracking the position and orientation of objects are exceptionally crucial in real-world scenarios. Most current methods rely on RGB-D-based approaches, such as BundleSDF [38], or 3D models for precise 6D pose estimation. However, these methods only function when depth information, point clouds, or CAD models are accessible. In certain extreme scenarios, depth information may be imprecise or absent, while RGB images remain stable. Therefore, it is critical to identify efficient ways to accurately estimate object poses and track them using only RGB images, which is a challenging and valuable pursuit.

Many algorithms in the literature, including [11, 14, 23, 44], exclusively use RGB images as input for object tracking, showing promising outcomes in terms of both speed and accuracy. Held et al. [14] present GOTURN, one of the first offline object trackers that can operate at 100 fps (*frame per second).*

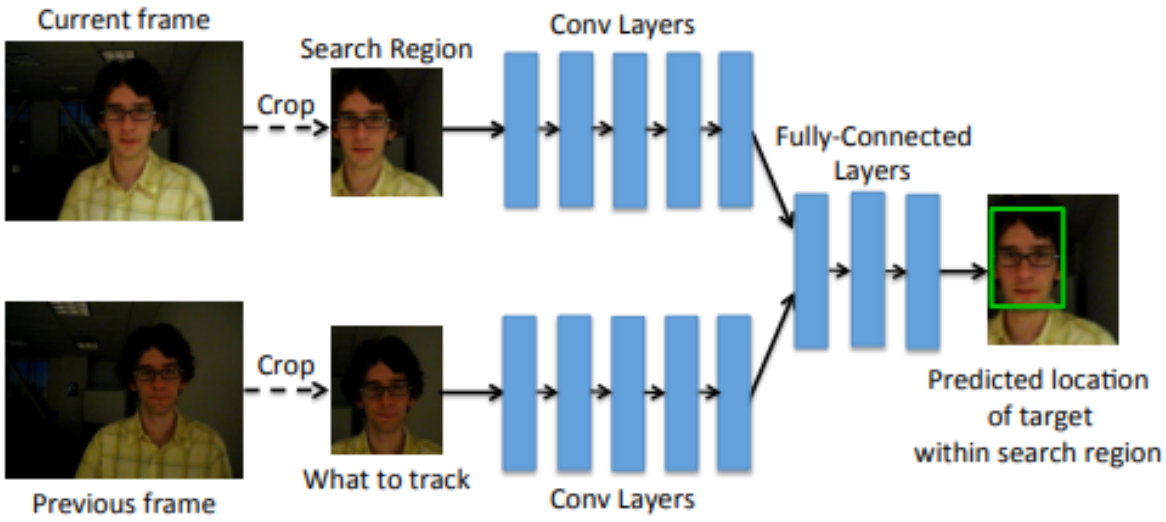


Figure 1.5: Architecture of GOTURN (from [14])

The tracker learns offline a generic correlation between an object's appearance and movement utilizing two basic chains of convolutional layers, followed by fully connected layers as shown in Figure 1.5. These layers compare the features of the target object with the characteristics in the current frame to detect the target's position after movement. The authors of [11] enhance the previous work by integrating a recurrent network ($LSTM$) post fully connected layers, introducing Re^3. This modification captures more intricate object transformations and maintains longer-term relationships than a standard convolutional network. As a result, this architecture improves accuracy and speed, enabling tracking of objects at 150 fps. Both models specialize in object position tracking and do not include pose tracking. Instead, tracking only the object pose is the specialization of [23] and [44]. MOTION-NETS, described in the first paper, is quite similar to the architecture of [11]: two branches for extracting features of the two images via convolutional layers, a 1x1 convolutional layer, an LSTM and fully connected layers. The second paper proposes CatTrack, a model that incorporates a temporal information capture module based on transformers to utilize the positional information of keypoints from the previous frame, and the PnP algorithm [9] to compute the poses.

There are limited studies available that present an architecture which can simultaneously track both position and pose.

1.3. 6 Degrees of Freedom Pose Representation

In computer vision, the 6 degrees of freedom pose is represented describing the position and orientation of an object in three-dimensional space. It involves six independent parameters that represent the object's movement and orientation relative to a reference frame. The 6 DoF are typically divided into two components:

1. **Translational degrees of freedom**: these three parameters define the object's position in the three-dimensional space along the X, Y, and Z axes.
2. **Rotational degrees of freedom**: these three parameters represent the object's orientation or rotation in 3-dimensional space around the X, Y, and Z axes.

The translation is often represented by a vector $\mathbf{t} = [t_x, t_y, t_z]^T \in \mathbb{R}^3$; the rotation, which belongs to the set

$$SO(3) = \{R \in \mathbb{R}^{3x3} | R^T R = I_{3x3},\ det(R) = 1\}, \tag{1.1}$$

instead can be described by several different parameterizations.

Common choices are Euler angles, unit quaternions, and rotation matrices.

Euler angles are a set of three angles used to represent the rotations around the three axes of a fixed coordinate system (X, Y, Z). They are referred to as:

- *Roll angle* (φ): rotation around the X-axis.
- *Pitch angle* (θ): rotation around the Y-axis
- and *Yaw angle* (ψ): rotation around the Z-axis.

By applying the rotations sequentially, the object's orientation can be represented uniquely. However, using Euler angles has some drawbacks, notably the phenomenon called "*gimbal lock*", which occurs when two of the angles align, leading to a loss of one degree of freedom. To avoid gimbal lock problems and provide better numerical stability, Euler angles are converted to quaternions or rotation matrices.

Quaternions are versors written in the following way:

$$\begin{aligned} q = w + ix + jy + kz,\ &where\ w, x, y, z \in \mathbb{R} \\ &with\ i^2 = j^2 = k^2 = -1 \\ &and\ w^2 + x^2 + y^2 + z^2 = 1. \end{aligned} \tag{1.2}$$

However, the representations with four or fewer dimensions for 3D rotation have discontinuities in Euclidean space.

Lastly, the **rotation matrix** is a square matrix 3x3 that can be written as:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{1.3}$$

To apply a rotation to a point $\mathbf{p} = [p_x, p_y, p_z]^T$, the resulting rotated vector $\mathbf{p}$' can be calculated using matrix multiplication:

$$\mathbf{p}' = R \cdot \mathbf{p} \tag{1.4}$$

Rotation matrices provide a concise and computationally efficient way to represent 3D rotations.

1.3.1. Projective Geometry

When we look around, the three-dimensional world becomes optically projected onto our retinas. The luminance is then detected by numerous photosensitive cells present in the retina. These measurements compile to produce an image that similarly resembles how a group of pixels on a screen forms an image. The optical principle of the human eye is identical to that of any optical camera, whether it is a photo or video camera. When projecting the 3D world onto the 2D retina, information regarding the 3D structure is lost.

One can model geometry in 3D space by utilizing the three-dimensional Euclidean vector space $\mathbb{R}^3$. For plane geometry (*images*), $\mathbb{R}^2$ is the equivalent choice. To represent a 2D point, a "fictitious" third coordinate is added to make it a 3D point. When dealing with projective geometry in 2D, we use **homogeneous coordinates**:

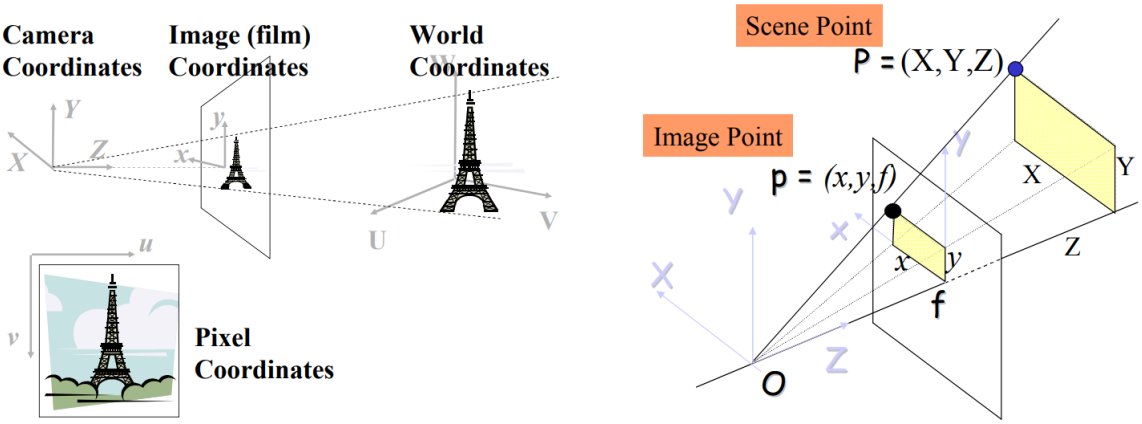
$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} \qquad \mathbf{p}' = k \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{1.5}$$

By convention, we can recover the 2D point $\mathbf{p}$ given the 3D point $\mathbf{p}$' as

$$x = \frac{x'}{z'} \qquad y = \frac{y'}{z'} \tag{1.6}$$

The initial step in transforming a 3D world point into a projected 2D pixel coordinates involves converting the world coordinates U, V, W (Figure 1.6a) into the camera coordinate system through a rigid transformation. This process comprises a rotation and a translation of the system. A 3D vector $\mathbf{T}$ represents the translation and describes the relative displacement between the origins of the two reference frames. Meanwhile, a 3x3 matrix R represents the rotation, aligning the axes of the two frames with each other. These parameters are known as the **extrinsic camera parameters**. To convert a point in the world frame ($\mathbf{P}_W$) to its counterpart in the camera frame $\mathbf{P}_C$, the following formula must be applied:

$$\mathbf{P}_C = R(\mathbf{P}_W - \mathbf{T}) \tag{1.7}$$



(a) Representation of the different coordinates systems [3]

(b) Representation of the perspective projection [3]

Figure 1.6: Geometry involved in computer vision

As illustrated in Figure 1.6b, if the real-world coordinates of point **P** in the camera system - X, Y, Z - are known, then it becomes feasible to compute the image coordinates on the image plane for the projected point **p**, which are x and y:

$$x = f_x \frac{X}{Z} \qquad y = f_y \frac{Y}{Z}, \quad where\ f = \begin{bmatrix} f_x \\ f_y \end{bmatrix}\ is\ the\ focal\ distance \tag{1.8}$$

The focal distance mentioned earlier is a parameter in matrix K that reflects the **intrinsic characteristics** of a camera's optical, geometric, and digital properties.

1.4. Challenges in 6D Pose Estimation and Object Tracking

Working on both pose estimation and object tracking algorithms presents numerous challenges. Firstly, it is easy to estimate and track an object in a simple environment, but this never happens in reality. In a real-world scenario, the target object will go through different difficulties, e.g. deformation, occlusion, background noise, etc. In this section, we explore some of the common issues that arise in 6D pose estimation and object tracking. In any machine learning or deep learning task, one of the main issues is caused by the background of the images. It is problematic to extract features, detect, esteem the pose, or even track an object when the **background** is **densely populated**: it introduces more redundant information and noise that makes the network less receptive to

important features.

In addition to the background clutter, another common challenge is the **occlusion of objects**. This phenomenon refers to a situation where multiple objects are so close to obscured one to each other or the background interferes with the target object causing the algorithms to mispredict the object's pose.

In a real-life scenario, the **illumination** on a target object changes drastically as the object moves making its localization more challenging to track and estimate.

Some challenges for the algorithms are caused by the type of object that is considered. They have problems if the object at issue is **symmetric** or **reflective**: in the first case, the object appears the same from multiple viewpoints, making it difficult for the model to determine its correct orientation or direction of movement; while, in the second case, the object can have an ambiguous appearance caused by lightning conditions and the angle of observation and lacks distinct features. One particular challenge faced by object tracking networks is the rapid motion of the object, which can significantly affect their ability to maintain accurate tracking across frames and images.

1.5. Unveiling some of the Core Datasets Driving Progress in 6D Pose Estimation and Object Tracking Research

Benchmarks are crucial for advancing research and development in augmented reality, as they offer a structured and measurable means of assessing the effectiveness and performance of diverse AR technologies and applications. Augmented reality is a multidisciplinary field with applications ranging from gaming and entertainment to education, healthcare, and industrial settings. The variety of potential applications and the complexity of the technology involved can make benchmarking and performance evaluation challenging. Different AR systems may be optimized for specific scenarios or tasks, making it crucial to have standardized benchmarks that cover a wide range of use cases. The majority of current algorithms rely on two main datasets for benchmarking: the *LineMOD dataset* [16] and the *YCB-Video dataset* [2]. The LineMOD dataset contains 15 objects, as illustrated in Figure 1.7, with low texture captured from different viewpoints in 15 videos, with each sequence featuring over 1,100 real images. However, only 13 of the 15 object sequences are used in the most recognized research.

Figure 1.7: LineMOD dataset objects

On the other hand, the original Yale-CMU-Berkeley (YCB) set includes 75 different objects selected to represent diverse scenarios and everyday objects with varying shapes, sizes, textures, weights, and rigidity. Xiang et al. [42] created the YCB-Video dataset that includes 21 out of the 75 original YCB objects (as shown in Figure 1.8). The dataset offers precise 6D pose annotations for 92 RGB-D videos with a total of 133,827 frames. The objects in the dataset possess various symmetries and are positioned in different poses and spatial arrangements, resulting in significant occlusions between them. As indicated in Table 1.1, we trained the model on 80 videos and reserved the remaining 12 for testing purposes. The sequences differ in the quantity of objects, ranging from a minimum of four to a maximum of nine. In the dataset, there are five symmetrical items (bowl, wooden block, large and extra-large clamp, and foam brick).

Number of objects	21
Total number of videos	92
Held-out videos	12
Min objects in a scene	3
Max object in a scene	9
Mean object in a scene	4.47
Number of frames	133,827
Resolution	640 x 480

Table 1.1: Statistics of YCB-Video dataset.

Figure 1.8: The subset of 21 YCB objects selected to appear in YCB-Video dataset (from [42])

2 | Proposed Method

In this chapter, we present a method intended to solve the problem of real-time inference of pose object estimation algorithm.

The objective of an object pose estimation method is to determine an object's three-dimensional position and orientation in space relative to a specific coordinate system: this involves obtaining detailed information about the object's translation and rotation. However, object pose estimation algorithms face a challenge of limited real-time inference. Several algorithms under this category are computationally intensive, leading to slower inference speeds. This limitation warrants significant consideration in contexts where real-time processing is crucial, e.g., robotics or interactive systems, and highlights the necessity for a judicious trade-off between accuracy and computational efficiency in method development. Our proposed approach seeks to surpass the limitations of balancing accuracy and computational efficiency by integrating a pose estimation model with a customized object tracking network to estimate object poses accurately and in real-time.

This chapter is divided into two parts: the first section provides a comprehensive outline of the proposed method, highlighting both its benefits and our objectives; the second section provides an in-depth description of our deep neural network object tracking. We begin the second section by describing the process of obtaining the two images that are used as inputs, and then presenting the network architecture for estimating the bounding box and pose. Subsequently, we present a simplified approach to let the network learn to forecast relative transformation, along with a description of the output regression losses.

2.1. Enhancing the Efficiency of Pose Estimation Networks for Real-Time Applications

In the realm of pose estimation algorithms, a formidable challenge arises from their considerable computational cost, leading to sluggish inference times that hinder their applicability in real-time scenarios. The compromise between prediction accuracy and real-time performance becomes necessary, posing a substantial obstacle in the field. In response,

our proposed approach addresses this critical issue by introducing a novel framework and refining existing methodologies, aiming to boost the efficiency of pose estimation networks without sacrificing accuracy.

The method proposed involves the integration of a pose estimation algorithm and an object tracking algorithm, leveraging the strengths of both neural networks. While pose estimation algorithms excel in accuracy, they often lag in speed during inference. On the other hand, object tracking models are designed for real-time applications with high inference speed but may compromise accuracy over time. Our innovative model, represented in Figure 2.1, strategically utilizes the pose estimation neural network at a lower frequency for accurate initial pose estimation. Simultaneously, at a higher frequency, it employs the object tracking model, called *SwiftTrack*, to track the object's pose in subsequent frames. This dynamic interplay ensures that the object tracking network continuously benefits from the pose estimation network's accurate predictions, avoiding delays and providing real-time performance.

GDR-Net is chosen as the pose estimation algorithm, with a new developed object tracking model, SwiftTrack, discussed in detail later. The proposed framework's flexibility allows for the incorporation of various pose estimation models, provided they output both the object's 2D bounding box and 3D pose.

2.2. SwiftTrack

The objective of object tracking is to locate and monitor the movements of a designated object or target with consistency over time.

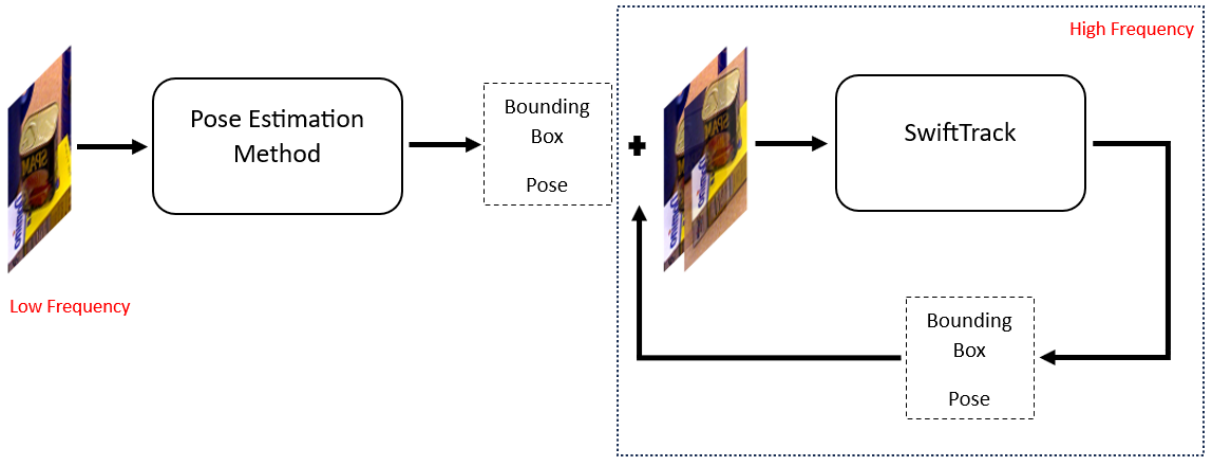


Figure 2.1: Structure of the proposed framework

To maintain their swift inference times, these models should avoid excessive size or complexity, as such attributes may compromise their efficiency. On the flip side, it's crucial for them to possess the capability to accurately estimate poses; otherwise, the risk of mispredictions in subsequent frames becomes pronounced. Bearing these considerations in mind, we present SwiftTrack, a novel object tracking model designed to estimate frame poses at an impressive speed of approximately 256 frames per second. SwiftTrack stands out for its innovative approach, overcoming size-complexity trade-offs and ensuring both speed and precision in pose estimation. Given two RGB images, one at time $t-1$ and another at time t, our network directly outputs the relative transformation needed to obtain the new 3D pose and object bounding box from the initial pose and bounding box.

2.2.1. Network Input

When a video contains multiple objects, the network needs to receive specific information about the tracked object. To accomplish this, only a portion of the image containing the target is inputted. A pair of crops from the image sequence is fed to the network at each frame. The first crop is centered on the object's location in the previous image, while the second crop is in the same location in the current image, as illustrated in Figure 2.2. This input enables our network to detect new objects that it has not previously encountered since the network tracks the object being input within these crops. To provide the network with contextual information concerning the surrounding area of the target object, we pad both crops with dimensions equal to the size of the object's bounding box incremented by 50%.

For instance, if the object's bounding box centers at

$$c^{t-1} = (c_x^{t-1}; c_y^{t-1}) \tag{2.1}$$

with width w^{t-1} and height h^{t-1} in frame $t-1$, both crops will be centered at c^{t-1} with width and height equal to $1.5w^{t-1}$ and $1.5h^{t-1}$, respectively. As long as the target is not occluded or moving too fast, it will remain in this region. However, for fast-moving objects, the search region may be expanded, which increases the network's complexity. Providing two crops to the network directly compares differences between two frames and learns the effects of motion on image pixels. Before they are input in the network, the crops are scaled to 64x64 pixels. The initial crop is necessary for the network to recognize the object and its starting position, while the second crop presents the search region to locate the object. The objective of the network is to determine the location of the target

object within the search region.

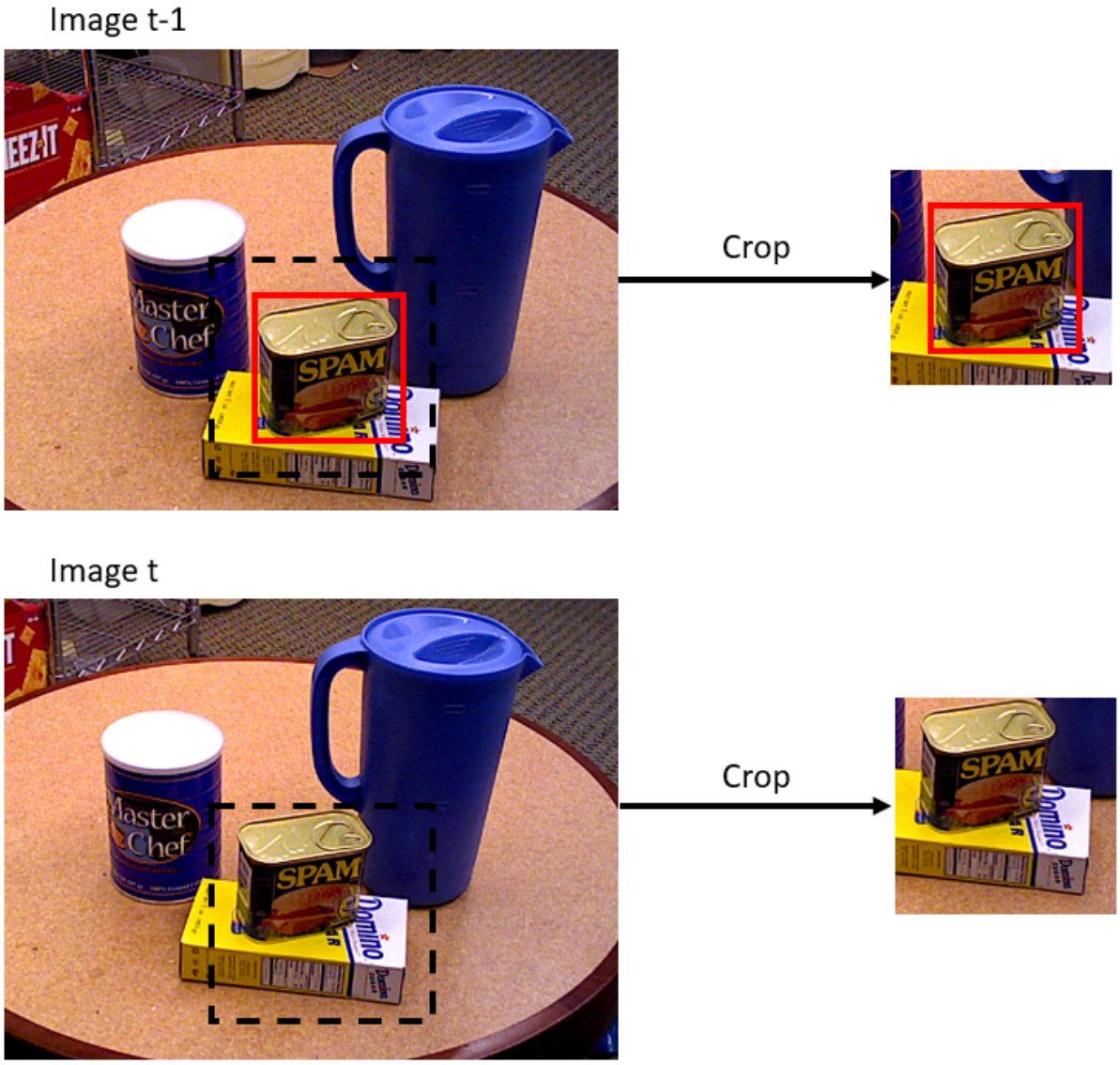


Figure 2.2: Example of cropped images at time $t-1$ and t of the YCB-Video dataset

2.2.2. Model Structure

Convolutional neural networks (*CNNs*) trained with backpropagation have demonstrated excellent performance in large-scale image classification and have also been widely applied in other areas such as semantic segmentation, depth prediction, edge detection, and keypoint prediction. CNNs are highly effective in learning input-output relations when sufficient labeled data is available. We thus adopt an end-to-end learning strategy for forecasting object bounding boxes and poses. To achieve this, we utilize a dataset comprising of image pairs and their corresponding ground truth positions.

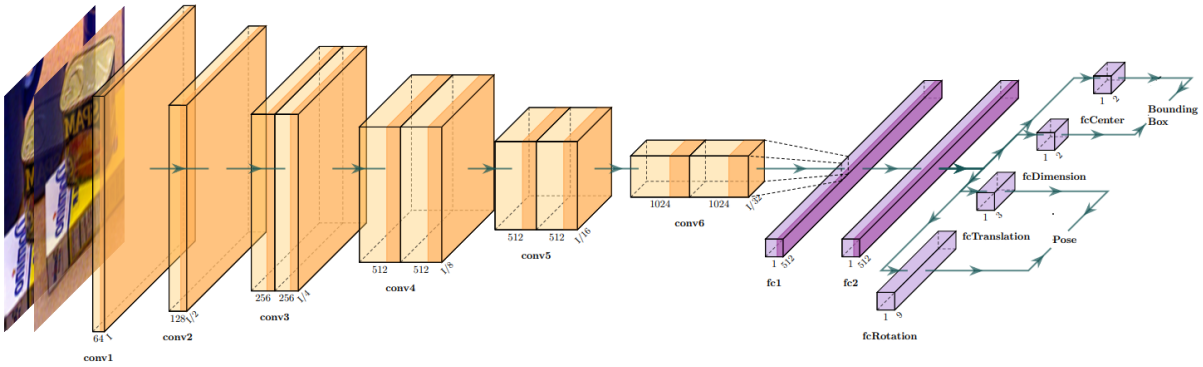


Figure 2.3: Network architecture for tracking.

We train a network to precisely predict the positions from the images. Figure 2.3 presents our network architecture utilized to track a single object within a scene. The two RGB images, which were cropped and scaled according to the instructions given in section 2.2.1, are concatenated to form a six-channel tensor input for the network (three channels for each image). The backbone network used is the FlowNetSimple architecture from Dosovitskiy et al. [8], trained to predict optical flow between two images. FlowNetSimple primarly consists of a succession of convolutional layers. The initial set of convolutional layers extracts feature representations, which capture details about the spatial and temporal variances present in both input images. The ultimate convolutional layers process the feature map and extract more intricate features associated with motion, while also predicting the optical flow vectors for every pixel in the input images. Our input tensor undergoes the first ten convolution layers of the FlowNetSimple. This processing generates a feature map that encapsulates valuable information pertaining to the motion of the object under consideration. These features are first flattened and then fed through two fully connected layers, each with a dimension of 512. Following this, four more fully connected layers are then used to predict the object's relative center transformation, as well as the target bounding box dimensions, translation vector, and rotation matrix.

In total, the model comprises about 39,5 M trainable parameters.

2.2.3. Structured Dissection of 3D Transformation Components

To reduce complexity and enhance network performance, we choose to employ a disentangled representation as described in [24]. By using this type of representation, we gain an advantage in which the individual components of relative transformations are maximally disentangled, thus eliminating the need for the network to learn intricate geometric relationships between translations and rotations.

The most common choice for representing object poses and transformations are camera coordinates. Given an initial object pose $[R_i|\mathbf{t}_i]$, the transformed final pose would be as follows:

$$R_f = R_\Delta R_i, \tag{2.2}$$

$$\mathbf{t}_f = R_\Delta \mathbf{t}_i + \mathbf{t}_\Delta, \tag{2.3}$$

where R_f and $\mathbf{t}_f$ indicate the final pose resulting from the transformation, R_Δ represents the relative rotation, and $\mathbf{t}_\Delta$ represents the relative translation. One problem with this representation is that the object rotation also impact the final translation due to the $R_\Delta \mathbf{t}_i$ term in Equation 2.2, even when the translation vector $\mathbf{t}_\Delta$ is zero. Another issue is that the network must remember the real size of each object to convert discrepancies in images to distance offsets. This is because the translation $\mathbf{t}_\Delta$ of an object is in the 3D metric space (for example, meters), which ties object size to distance in the metric space. An example can be observed in column (b) of Figure 2.4. The upper row displays the rotation center at the image's center, leading to an unwanted translation alongside the object's rotation. Additionally, in the bottom row, one can note the additional translation necessary to offset the translation caused by the rotation. To overcome these problems, Li et al. [24] suggests decoupling the estimation of R_Δ and $\mathbf{t}_\Delta$.

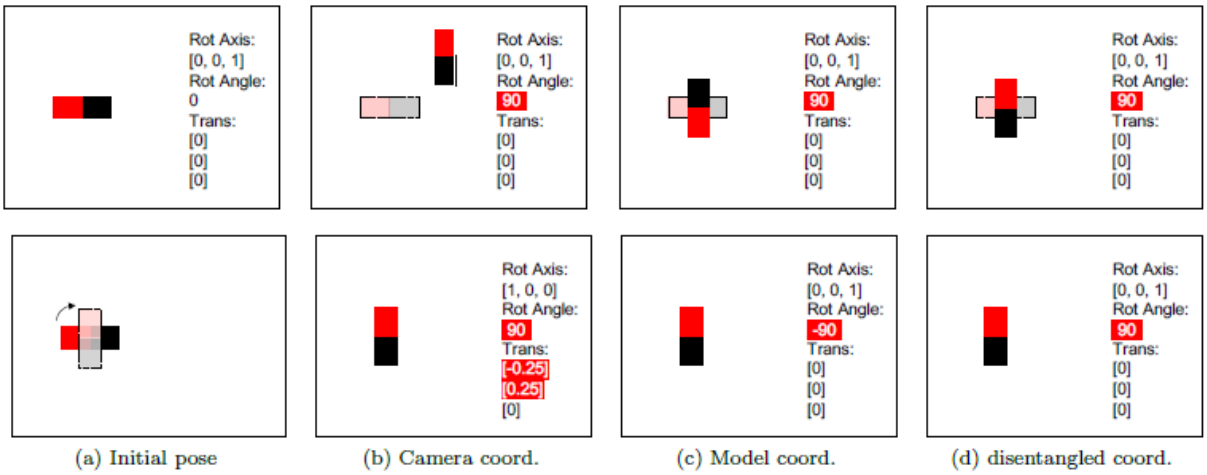


Figure 2.4: Behavior of the rotation of an object in different reference systems (from [24]). The upper row panels demonstrate the impact of a 90-degree rotation along the image plane axis on the object displayed in column (a). The bottom row displays the rotation vectors that a network ought to provide to achieve in-place rotation using the different coordinate systems.

The initial step is to shift the center of rotation from the camera's origin to the object's center in the camera frame, as indicated by the current pose estimate. This adjustment ensures that object translation in the camera frame does not change upon rotation. The next stage involves employing axes that are parallel to those of the camera frame when calculating the relative rotation. By using this method, the network can be trained to estimate the relative rotation independently of the coordinate frame of the 3D object model. This is illustrated in column (d) of Figure 2.4. This axes configuration is chosen because if the axes were specified in the 3D object model, the network would be required to learn and memorize the coordinate frames of each object in order to determine the correct rotation axis and angle. This would make the training process more difficult. This concept is demonstrated in column (c) of Figure 2.4. In this example, the object model can be defined arbitrarily, allowing for rotation along any axis with the same rotation vector. Despite the clockwise rotation shown in the image, the same axis could also result in an out of plane rotation if the object coordinate frame is defined differently.

Translation Estimation from 2D Images

Based on the previous section's discussion of separating transformation components, we now describe our method for predicting three-dimensional translation.

A clear method for representing relative translation is

$$\begin{aligned} \mathbf{t}_\Delta &= (\Delta_x, \Delta_y, \Delta_z) = \mathbf{t}_f - \mathbf{t}_i, \\ where\, \mathbf{t}_f &= (x_f, y_f, z_f)\, and\, \mathbf{t}_i = (x_i, y_i, z_i). \end{aligned} \tag{2.4}$$

Estimating relative translation in the 3D metric space using only 2D images lacking depth information poses a challenge for the network. The network must identify the object's size and translate its movement from 2D space to 3D based on that size. Rather than directly training the network to regress the vector in 3D space, we regress to the changes in the object's positioning on the 2D image plane. In particular, we train the network to regress to the relative translation $\mathbf{t}_\Delta = (v_x, v_y, v_z)$, where v_x and v_y represent the number of pixels the object should move along the image x-axis and y-axis, respectively, and v_z pertains to the object's scale modification. To derive vector $\mathbf{t}_\Delta$, we combine Equation 1.6 and Equation 2.4 to obtain the following relationship:

$$
\begin{aligned}
v_x &= f_x(\frac{x_f}{z_f} - \frac{x_i}{z_i}), \\
v_y &= f_y(\frac{y_f}{z_f} - \frac{y_i}{z_i}), \\
v_z &= log(\frac{z_i}{z_f}),
\end{aligned}
\tag{2.5}
$$

where f_x and f_y are the focal lengths of the camera

The scale change v_z is defined as independent of the absolute object size or distance by utilizing the ratio between the distances of the two observed images. The logarithm utilized to compute v_z guarantees sure that a value of zero denotes no alteration in scale or distance.

Thanks to the utilization of Equation 2.5, the network has the capability to predict the novel translation vector $\mathbf{t}_f$ of the object in the recently allocated position.

Rotation Estimation

There are multiple well-known approaches to illustrate general SO(3) rotations for deep models, as discussed in Section 1.3. Nevertheless, we can expect that the motion between two successive frames will be relatively small, and thus our representation should focus on small rotations. As a result, we implement the gnomonic projection as discussed in [13]. This method utilizes a quaternion sphere S^3, which is represented in the four-dimensional space $\mathbb{R}^4$, to project points from a three-dimensional sphere onto three-dimensional space $\mathbb{R}^3$. The projection originates from the sphere's center, (0,0,0,0), and extends to a tangent hyper-plane in $\mathbb{R}^3$, a plane that touches the sphere at a single point, as illustrated in Figure 2.5. The gnomonic projection is a 2-to-1 mapping of S^3, meaning that opposite points on the sphere are projected to the same point in $\mathbb{R}^3$. However, our network does not use quaternions but instead employs rotation matrices. This is not problematic because, as is explained in [13], there is an equivalent mapping of the gnomonic projection called Cayley transform. The Cayley transform on matrices is defined as the mapping

$$
A \rightarrow A^C = (I - A)(I + A)^{-1}. \tag{2.6}
$$

This mapping is valid for any square matrix as long as $(I + A)$ is invertible. In addition, the Cayley transform is its own inverse, so $(A^C)^C = A$.

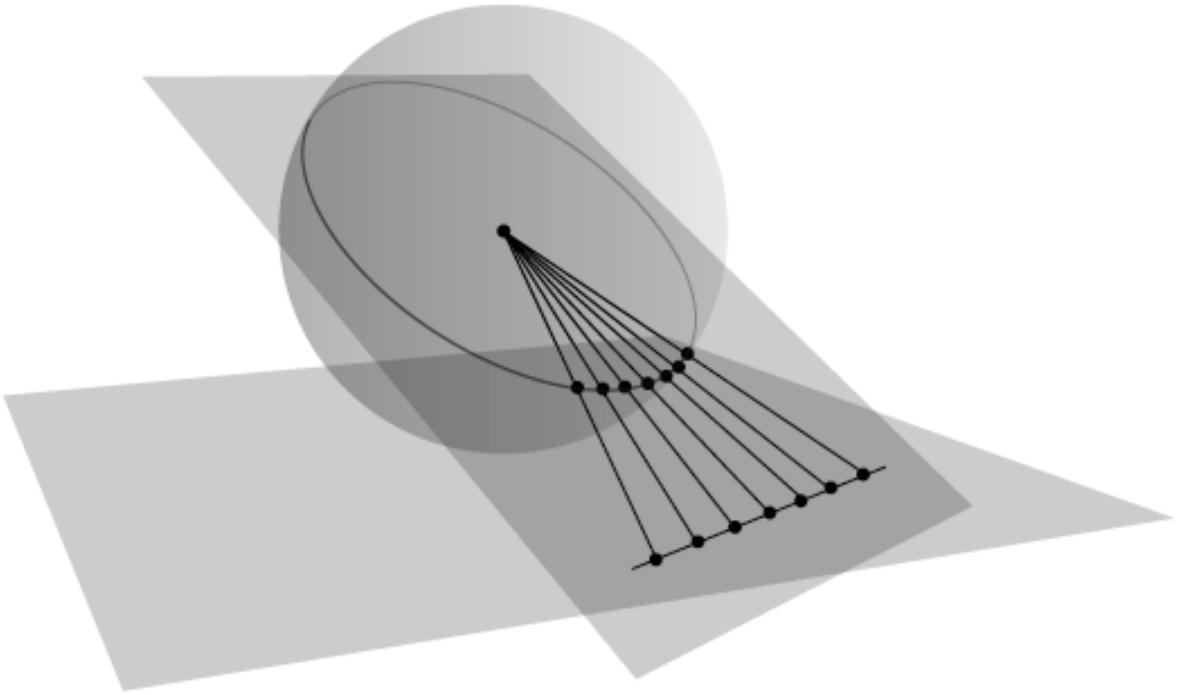


Figure 2.5: Gnomonic projection of a sphere (from [13]).

Proposition 2.1. *The Cayley transform of a rotation matrix $R \in SO(3)$ is a skew-symmetric matrix, and vice versa. Thus the correspondence $R \overset{C}{\longleftrightarrow} [\mathbf{v}]_\times$ is a one-to-one correspondence between skew-symmetric matrices and rotations R, excluding rotations through an angle of π.*

This proposition highlights two primary benefits of the transformation:

- It is highly effective for infinitesimal rotations because it yields a skew-symmetric matrix.
- A one-to-one correlation exists between skew-symmetric matrices and 3D space rotation, meaning that each skew-symmetric matrix represents a distinct rotation.

The only disadvantage is that this correspondence does not hold for angles of π (180 degrees). The importance of the Cayley transform in rotations is extensively explained in [39], where Wu demonstrates that this transformation is an effective technique for improving the numerical stability of algorithms.

Once the relative rotation is obtained, Equation 2.2 is used to calculate the final rotation of the object.

In conclusion, this method of representing relative transformation has numerous advan-

tages:

1. Rotation no longer affects the estimation of translation, which eliminates the need to offset the movement caused by rotation around the camera center.
2. The intermediate variables of v_x, v_y, and v_z represent basic translations and scaling alterations in the image plane.
3. This representation does not need any previous knowledge of the object.

2.2.4. Object Detection and Pose Estimation Loss

The network outputs four elements: the center and dimensions of the bounding box, the translation vector, and the rotation matrix of the object in the frame at time t. If we consider this configuration, the total network loss is

$$L_{network} = \alpha L_{center} + \beta L_{wh} + \gamma L_{trans} + \delta L_{rot}. \tag{2.7}$$

To calculate how far the various predictions are from the target, we can use the L_1 distance for all four measurements:

$$L_*(\mathbf{y}, \hat{\mathbf{y}}) = |\hat{\mathbf{y}} - \mathbf{y}|. \tag{2.8}$$

However, employing four distinct loss functions for the four outputs results in difficulties in balancing the various losses. This is due to the fact that the four components are on different scales. The component error associated to the bounding box are calculated in pixels, while translation and rotation are measured in millimeters and radians, respectively. Thus, we opt to merge the components two by two into two separate losses to address partially this issue:

$$L_{network} = \alpha L_{bbox} + \beta L_{pose}. \tag{2.9}$$

Again, the problem of balancing losses arises, but a loss with four components is certainly more challenging and less explored than a loss with two components. Additional details on this topic are elaborated on later in the thesis. Due to the distinct characteristics of the bounding box and pose, developing a single loss that covers all four outputs was not feasible.

Unified Bounding Box Loss Calculation

Since we have the center coordinates $c = (c_x; c_y)$ and the dimensions w, h of the bounding box and aim to merge them into a single loss, we compute the upper left corner coordinates and lower right corner coordinates to obtain the final predicted bounding box $\hat{\mathbf{b}}$ as follow:

$$\hat{\mathbf{b}} = \begin{bmatrix} x_{left} \\ y_{up} \\ x_{right} \\ y_{down} \end{bmatrix} = \begin{bmatrix} c_x - \frac{w}{2} \\ c_y - \frac{h}{2} \\ x_{left} + w \\ y_{up} + h \end{bmatrix} \tag{2.10}$$

Subsequently, we determine the mean absolute error between the predicted and target bounding boxes through the utilization of $L1$ loss, also known as Absolute Error Loss:

$$L_{bbox}(\mathbf{b}, \hat{\mathbf{b}}) = \frac{1}{N} \sum_{n=1}^{N} |\hat{\mathbf{b}}_n - \mathbf{b}_n^t|, \;\; \textit{where } N \textit{ is the batch size.} \tag{2.11}$$

Unified Pose Regression Loss

To create a unified loss function for pose regression that combines the translation vector and rotation matrix, we employ the geometric reprojection error introduced by Kendall and Cipolla et al. [19]. This loss calculates the mean distance between the 2D projections of 3D points in the scene, using both the estimated pose and the ground truth pose. Given the true pose of an object, represented as p = [R|**t**], and the estimated pose, represented as $\hat{p} = [\hat{R}|\hat{\mathbf{t}}]$, the loss is calculated as:

$$L_{pose}(p, \hat{p}) = \frac{1}{n} \sum_{i=1}^{n} ||(\hat{R}\mathbf{x}_i + \hat{\mathbf{t}}) - (R\mathbf{x}_i + \mathbf{t})||_2, \tag{2.12}$$

where $\mathbf{x}_i$ represents a randomly selected 3D point on the object model and n is the total number of points. As our dataset only includes the real dimensions of each object, 3D bounding boxes are calculated, resulting in $n = 8$. This formulation is also used to compute the average distance (ADD) metric in Equation 4.4.

3 | Preliminary Analysis on Embedded Platforms Timing

One of the most important aspects related to this research is performance. For this reason first of all we have conducted an investigation of how the algorithms proposed in Section 1.1.2 perform. In particular, in this chapter we analyze and discuss the performance obtained by different embedded devices when performing 6D pose estimation.

The chapter begins with a general introduction to the two algorithms under study, outlining the rationale for our selection, and our overall goals. In the following section, we conduct a comparative analysis of several devices from the *NVIDIA Jetson* linup.

We explain their technical features, identify potential issues encountered during tests, and present their performance in predicting pose estimation. In the final section, we conduct a comparable analysis that includes a standard central processing unit (*CPU*) and a graphics processing unit (*GPU*).

3.1. Diving into GDR-Net and DenseFusion Models

In Section 1.1.2, we extensively examined the architecture of two 6D pose estimation models, namely GDR-Net and DenseFusion. The key distinction that we find relevant between these algorithms is in the nature of their input. Specifically, GDR-Net is a neural network built for processing RGB images, while DenseFusion utilizes both RGB images and depth information to estimate the pose of an object. We examine these two models to evaluate whether the input given to the network has a significant impact on timing and accuracy. The goals of this investigation are:

- Assessing whether there is a significant decrease in performance when using a standard RGB camera compared to an RGB-D camera in a practical application.
- Estabilishing whether using lower-performing hardware results in a significant performance decrease in inference times on embedded systems. Such hardware, though,

is ideal for real-world applications with limited available space.

	GDR-Net (*MiB*)	DenseFusion (*MiB*)
Memory usage	834	766

Table 3.1: Memory used by GDR-Net and DenseFusion models.

As shown in Table 3.1, the GDR-Net model occupies more memory than DenseFusion.

Both neural networks perform accuracy and inference calculations on the LineMOD dataset. This dataset consists of 15 objects, as explained in Section 1.5. However, during model training or testing, only 13 of these objects are utilized. Table 3.2 illustrates the 13 objects included in the dataset and provides insight into the accuracy of each model's estimation of the 6D pose for each object. As can be seen, the DenseFusion model achieves the highest average accuracy of 95.23%, achieving an astonishing level of accuracy exceeding 98% in objects such as *eggbox*, *glue*, and *iron*.

	Accuracy (%)	
	GDR-Net	**DenseFusion**
Ape	76.29	92.76
Benchvise	97.96	93.89
Camera	95.29	96.57
Can	98.03	94.00
Cat	93.21	96.61
Driller	97.72	87.91
Duck	80.28	93.05
Eggbox	99.53	99.81
Glue	98.94	99.81
Holepuncer	91.15	92.48
Iron	98.06	98.16
Lamp	99.14	96.83
Phone	92.35	96.06
Average	*93.69*	*95.23*

Table 3.2: Accuracy of the GDR-Net and DenseFusion models on the 13 LineMOD objects.

However, GDR-Net also performs well and even surpasses Densefusion's accuracy for certain objects like *benchvise*, *can*, and *lamp*.

Moving forward, we conduct a thorough analysis of time inference on various embedded platforms. In addition to their disparate inputs, DenseFusion and GDR-Net exhibit another difference. Specifically, DenseFusion performs an inference step to estimate the object's pose and a subsequent post-processing step to refine it. On the other hand, GDR-Net solely executes the inference step. As a result, the post-processing row for GDR-Net in all subsequent tables reads "−".

3.2. Comparison between CPU and GPU

The Central Processing Unit and Graphics Processing Unit are the first hardware components used to test the two models. Both the CPU and GPU are microprocessor types, yet they have distinct functions. CPUs execute instructions and are optimized for serial processing, enabling swift task switching while handling a single task at a time. On the other hand, GPUs are specialized processors designed to handle complex mathematical calculations and render graphics. They are optimized for parallel processing, enabling them to handle multiple tasks simultaneously.

To deploy them, we utilize Docker [7], an open platform for building, transmitting, and executing applications. Docker enables the packaging and execution of applications in a loosely isolated environment, commonly referred to as a *container*. Docker allows for the segregation of applications from their underlying infrastructure, facilitating expedited delivery of software. This feature permits the utilization of the Dockerfile included in the DenseFusion repository to construct an image that is utilized by the container to execute its code. DenseFusion, released in 2018 with PyTorch version 0.4.1, presents some issues nowadays as few of its functions are deprecated, so we adjust the code to ensure functionality in an environment with a PyTorch version greater than 1.0.

On the other hand, the GDR-Net repository does not contain a Dockerfile ready to be used, necessitating the need to craft one from scratch considering all the different requirements and compatibilities of packages necessary to create a proper environment to run the model. Subsequently, we utilize four CPUs for each model to predict the object's pose estimate from the acquired image. Using the time library in Python, we can compute the duration required by the models to determine the position for each image in the LineMOD test dataset. At the conclusion of the inference process, we can store those values in a CSV file. Subsequently, we import the CSV file into Excel to assess the average and variance of the values over the entire dataset. The average inference time is converted into fps using

	GDR-Net	DenseFusion
Average Inference (*s*)	0.11192	0.16817
Average Post-processing (*s*)	–	0.03675
Total Average Inference (*s*)	*0.11192*	*0.20492*
Total Inference Variance (*s*)	*0.00002*	*0.00443*
Total Average Inference (*fps*)	*9*	*5*

Table 3.3: Time inference calculated using 4 CPUs.

the formula

$$T_{fps} = \frac{1}{T_s}, \tag{3.1}$$

where T_s represents the time in seconds. The total average inference of both algorithms is shown in Table 3.3, proving a time of 9 fps for GDR-Net and 5 fps for DenseFusion.

After conducting experiments using CPUs, we employed graphics processing units, specifically the NVIDIA GeForce GTX 1080 for GDR-Net and the NVIDIA RTX 6000 for DenseFusion, to conduct further tests. Again, we apply the identical method to determine the times as previously outlined for the CPU. As indicated in Table 3.4, both models experienced significant time reductions compared to CPU calculations. This is reasonable since GPUs are specifically designed for executing complex mathematical operations simultaneously. From the table, we can see that GDR-Net operates at approximately 70 fps, while DenseFusion operates at around 52 fps.

	GDR-Net	DenseFusion
Average Inference (*s*)	0.01447	0.00959
Post-processing (*s*)	–	0.00947
Total Average Inference (*s*)	*0.01447*	*0.01907*
Total Inference Variance (*s*)	*0.00019*	*0.00010*
Total Average Inference (*fps*)	*70*	*52*

Table 3.4: Time inference calculated using GPU.

3.3. NVIDIA Jetson Developer Boards

NVIDIA Jetson [28] comprises embedded computing platforms designed to enhance artificial intelligence (*AI*) and deep learning applications at the edge with acceleration. NVIDIA, a renowned technology corporation famous for its GPUs and AI innovations, developed these platforms. Jetson devices are compact, energy-saving, and formidable, rendering them suitable for implementing artificial intelligence and computer vision applications in a range of settings where real-time processing and inference are vital. Jetson devices encompass NVIDIA's GPU technology, which accelerates AI workloads, allowing them to function effectively and execute intricate tasks, such as object detection, image recognition, and robotics control. Jetson platforms are equipped with software libraries, tools, and frameworks that facilitate the development of AI-powered applications. The NVIDIA Jetson series comprises several widely used devices such as *Jetson Nano*, *Jetson Xavier NX*, *Jetson AGX Xavier*, and *Jetson TX2*.

The thesis compares the performance of the t models on Jetson Nano, Jetson AGX Xavier, and Jetson TX2 devices. The performance of each module is analyzed in detail in the subsequent sections.

3.3.1. NVIDIA Jetson Nano

The initial embedded platform selected from the Jetson series is the NVIDIA Jetson Nano, displayed in Figure 3.1. The Jetson Nano is a cost-effective and portable single-board computer (*SBC*) designed for AI and machine learning purposes. It includes a quad-core ARM Cortex-A57 CPU that can handle ordinary computing tasks, in tandem with a Maxwell architecture GPU equipped with 128 CUDA cores for improving the efficiency of AI and deep learning operations. It efficiently multitasks due to the 4GB of LPDDR4 RAM. The Jetson Nano employs a customized Linux-based operating system, known as NVIDIA JetPack, that's specifically optimized for machine learning tasks while providing essential software libraries and tools for development. It provides numerous connectivity options, such as USB ports and GPIO pins, making it ideal for interfacing with external devices and sensors. The compact form factor, copious documentation, and robust developer community support of this SBC make it a favored option for a diverse array of projects, spanning from robotics and smart cameras to Internet of Things (*IoT*) and educational applications. This empowers users to explore and implement real-time AI solutions at the edge with ease.

Figure 3.1: NVIDIA Jetson Nano

Table 3.5 displays the inference times acquired from two deep learning models on the Jetson Nano using the same process described in the previous section. In this instance, GDR-Net achieves inference speeds of approximately 7 fps, whereas DenseFusion operates at around 5 fps. Acquiring these results proved to be quite challenging, particularly during our attempts to execute GDR-Net. As previously explained in Section 3.2, we relied on Docker to develop ad hoc environment for running our algorithms. This was possible due to the availability of both Dockerfiles. The process of generating the Docker container for DenseFusion was relatively unproblematic. The sole obstacle encountered concerned the incompatibility of the CPU's ARM architecture with the .so file (short for *Shared Object*) contained within the original repository, which was designed for an x86_64 architecture. Once the new .so file was created, inference times could be calculated on the LineMOD dataset without encountering any further issues. In contrast, the issue encountered during the GDR-Net project was dissimilar. The project's requirements state that GDR-Net runs on Python versions greater than 3.5, which is compatible with the latest JetPack version for the Jetson Nano that utilizes Python 3.6. However, numerous features of Detectron2 [41], Facebook AI Research's advanced library that offers up-to-date detection and segmentation techniques, are utilized in this project, necessitating a minimum of Python version 3.7.

	GDR-Net	DenseFusion
Average Inference (*s*)	0.15013	0.14918
Post-processing (*s*)	–	0.05517
Total Average Inference (*s*)	*0.15013*	*0.20435*
Total Inference Variance (*s*)	*0.05718*	*0.10773*
Total Average Inference (*fps*)	*7*	*5*

Table 3.5: Time inference calculated using NVIDIA Jetson Nano.

This has posed a challenge as there have been no JetPack releases beyond 3.6 for the Jetson Nano. Fortunately, a hardware-specific image [29] was discovered, featuring Ubuntu 20.04 and updated editions of Python, PyTorch, and TorchVision. The microSD card was flashed with the image utilizing a dedicated software and reinserted into the Jetson Nano. Following a few concluding system configurations, all requisite packages for the algorithm's correct operation were installed.

3.3.2. NVIDIA Jetson TX2

The NVIDIA Jetson TX2, shown in Figure 3.2, is a member of the Jetson family of powerful and feature-rich single-board computers designed for high-performance edge computing and embedded AI applications. It boasts a formidable six-core CPU comprising a dual-core ARM Cortex-A57 and a quad-core ARM Cortex-A53, in addition to a Pascal architecture GPU with 256 CUDA cores, delivering ample processing power for AI and deep learning tasks. With 8 GB of LPDDR4 RAM and 32 GB of eMMC storage, the Jetson TX2 efficiently handles complex workloads. Although slightly larger than the Jetson Nano, the Jetson TX2 is also a compact SBC. Like all hardware in the Jetson series, it offers numerous connectivity options and runs on the customized Linux-based NVIDIA JetPack operating system. Its combination of a potent CPU, GPU, and sufficient RAM make it ideal for intricate AI and computer vision tasks. It has gained recognition in professional and research applications that demand more computational power when compared to the Jetson Nano.

For this particular hardware, the only available data on performance pertains to the DenseFusion project, with a recorded speed of approximately 6 fps. As shown in Table 3.6, the inference row of GDR-net displayed a value of "−", which we attribute to incompatibility between the Jetson and project packages, hindering our ability to run the

model effectively. NVIDIA has not released versions above 5.0 of the JetPack containing Python above 3.7 for this device, as we anticipated in Section 3.3.1 for the Jetson Nano. Due to installation errors with Detectron2 and other software packages, such as OpenCV, the creation of a necessary Docker image for running GDR-Net was unsuccessful. Despite our efforts, the problem remained unresolved, leading us to ultimately abandon the possibility of obtaining inference times for this project on this device. On the other hand, we encountered no difficulties and accomplished our intended outcomes for DenseFusion, owing to the established procedure of constructing the Docker ecosystem employed in the Jetson Nano.

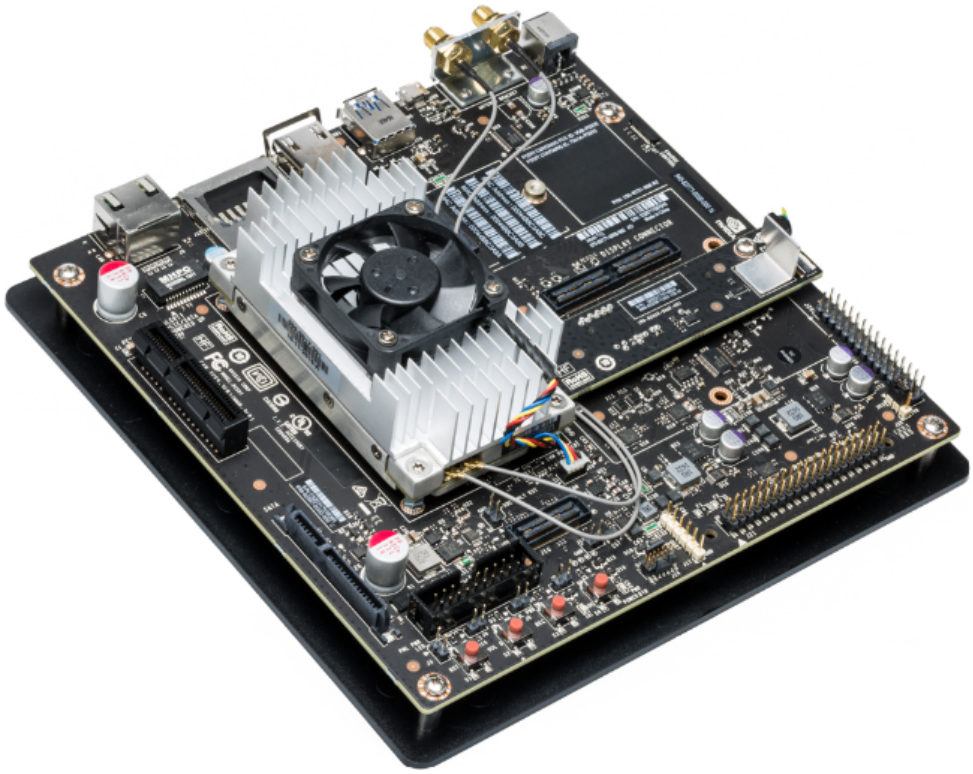


Figure 3.2: NVIDIA Jetson TX2

	GDR-Net	DenseFusion
Average Inference (*s*)	–	0.09965
Post-processing (*s*)	–	0.05770
Total Average Inference (*s*)	–	*0.15735*
Total Inference Variance (*s*)	–	*0.00834*
Total Average Inference (*fps*)	–	*6*

Table 3.6: Time inference calculated using NVIDIA Jetson TX2.

3.3.3. NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier, displayed in Figure 3.3, represents the peak of the Jetson line of single-board computers, delivering outstanding computational power and AI features. It features a custom eight-core CPU melding ARM Cortex-A57 and Cortex-A53 cores, accompanied by a Volta architecture GPU containing 512 CUDA cores that offer remarkable GPU acceleration for AI and deep learning workloads. It is capable of competently handling even the most intricate assignments thanks to 16GB of LPDDR4x RAM and 32GB of eMMC storage. The board supports well-known AI frameworks and provides significant hardware acceleration, making it fitting for challenging neural network inference tasks. Its compact size and connectivity choices enable it to be included in a range of uses, such as robotics, self-governing automobiles, medical imaging, industrial automation, and sophisticated research projects. NVIDIA's abundance of developer resources, such as those available for Jetson Nano and Jetson TX2, make it an excellent candidate for professionals and researchers actively involved in AI and computer vision endeavors on the front lines, where top-notch performance remains of utmost importance.

Figure 3.3: NVIDIA Jetson AGX Xavier

Calculating the inference times of both algorithms posed no difficulties on this hardware. The device already had JetPack 4.6.4 with embedded Python 3.6 installed; thus, we utilized it to execute DenseFusion as it complied with all necessary prerequisites. Moreover, we could install all the packages and libraries required to operate the model properly, thanks to the Dockerfile and the established process.

	GDR-Net	**DenseFusion**
Average Inference (*s*)	0.04025	0.02625
Post-processing (*s*)	–	0.02071
Total Average Inference (*s*)	*0.04025*	*0.04696*
Total Inference Variance (*s*)	*0.00468*	*0.00071*
Total Average Inference (*fps*)	*25*	*21*

Table 3.7: Time inference calculated using NVIDIA Jetson AGX Xavier.

After obtaining the inference times of DenseFusion, we upgraded the JetPack on the Jetson AGX Xavier to the newest version featuring Python 3.8. This was necessary to allow GDR-Net to run due to the challenges discussed in detail in Sections 3.3.1 and 3.3.2. Table 3.7 shows that GDR-Net has an inference speed of approximately 25 fps, while DenseFusion achieves a speed of 21 fps.

3.4. Final Remarks

In this chapter, we compared the accuracy and timing performance of GDR-Net and DenseFusion algorithms for object pose estimation on various embedded platforms.

DenseFusion achieves high accuracy by estimating a good pose for most objects in the dataset and utilizes less memory compared to GDR-Net. However, GDR-Net, while having good average accuracy, is unbalanced in its performance, showing high accuracy for some objects but not for others. Looking at the inference time results obtained by each individual device (CPU, GPU, Jetson Nano, Jetson TX2, and Jetson AGX Xavier), it is evident that GDR-Net achieves faster inference times than DenseFusion. Also, as expected, the best timings are obtained using a GPU, and inference times improve whenever a Jetson device with better hardware specifications is used. Finally, it is evident that the inference timings are very similar when using a CPU and a Jetson Nano, so if we perform an inference analysis on one of the two devices, we can get an idea of what the timings will be on the other hardware.

This analysis shows that the best times are obtained by using GDR-Net, an algorithm that only takes RGB images as input, but with the consequence that you have a decrease in accuracy. In addition, depending on the required speed of inference for a real-world application, the optimal embedded platform can be determined.

4 | Experiments and Results

In this chapter, we provide a detailed account of the experiments conducted to arrive at the final model. We also address the encountered problems during the algorithm's development and the solutions we implemented.

The first section elaborates on the dataset used for training, including the restructuring process to satisfy our specific needs and how we tackle the associated challenges. In the following section, we address the general implementation of the model and the adjustment of hyperparameters along with other technical details. After that, we elaborate on the performance metrics utilized to evaluate the algorithm's efficiency. Lastly, the subsequent section presents the executed experiments followed by a comprehensive analysis of the acquired results.

4.1. Utilizing YCB-Video Dataset for Model Training and Evaluation

YCB-Video is the dataset chosen to train and evaluate the model proposed in this research. As mentioned in Section 1.5, the YCB-Video dataset includes 21 objects, represented in Figure 1.8, characterized by their size and texture, and comprising both real-world and synthetic images. The real-world folder of the YCB-Video dataset contains 92 videos filmed using an RGB-D camera in diverse locations. We follow prior investigations and use 80 videos for training while designating the remaining 12 for testing. We select 15 out of the 80 training videos randomly to create a validation dataset, which constitutes 20% of the total. These 15 videos are separated from the training dataset to ensure they are not used for training. On average, each video contains 1,500 frames with 4 to 9 observable objects. Hence, when creating the customized training dataset, we crop each image of the 65 videos for each object, resulting in over 400,000 samples. We apply the same procedure to the validation dataset but obtain a smaller subset with over 70,000 samples due to only considering 15 videos in this case. It is evident that the number of samples in a dataset is considerably high, resulting in a significant increase in the time required

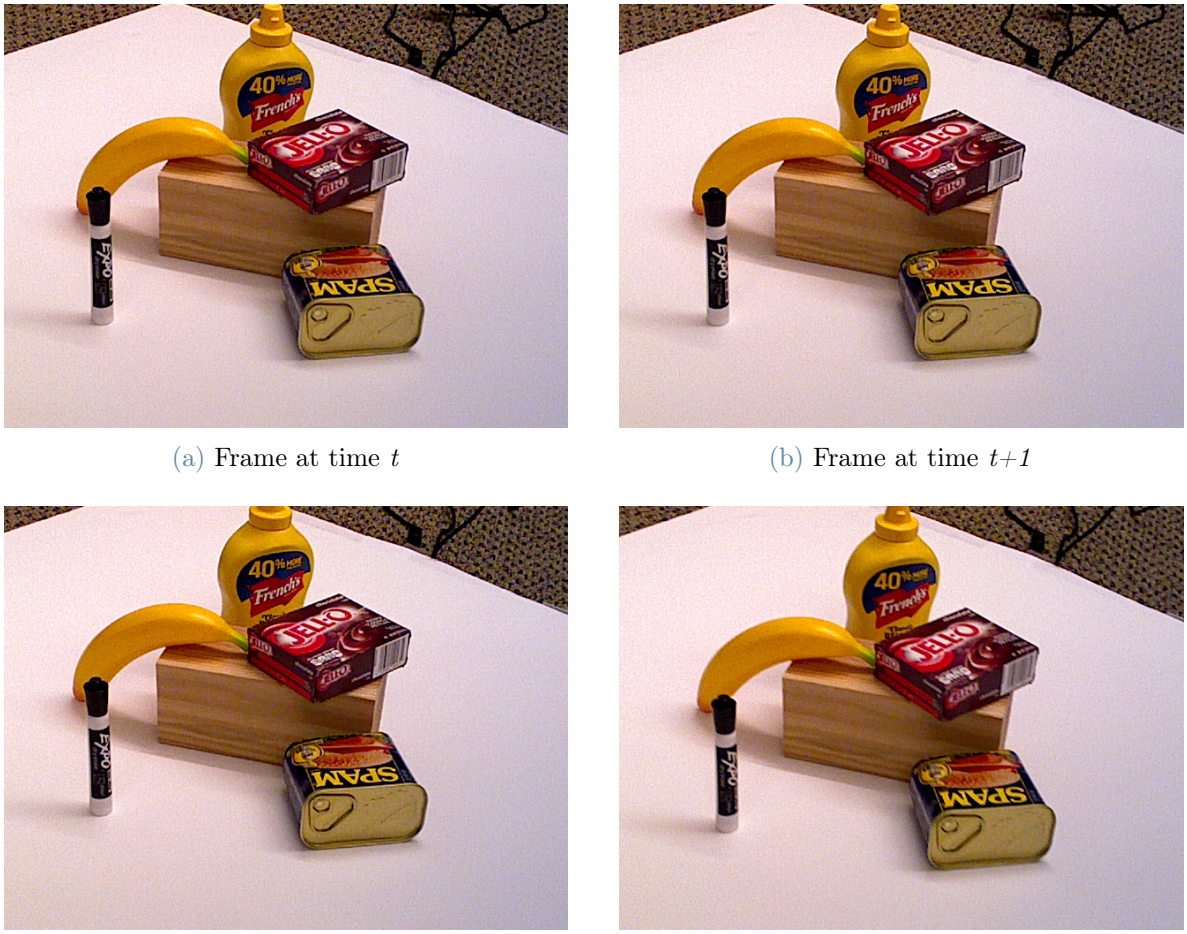
for model training, with a conclusion reached only after 48 hours of training. To expedite our progress and reduce time spent, we opt to minimize our sample sizes. To achieve this goal, the data pre processing algorithm randomly select only 100 frames from each image sequence of the videos, which is used for both the training and validation datasets. To ensure that all subsequent experiments can be compared with one another, it is imperative that the 100 random frames selected for each video are consistent throughout. Therefore, we generate a JSON file that includes the ID numbers for both the selected validation dataset videos and the 100 chosen random frames. The training set, thus, comprises 28,954 samples, while the validation set comprises 6,459 samples.

In addition, the YCB-Video dataset comprises approximately 80,000 synthetic images utilized to create 3D models of objects within the simulated environment. Furthermore, the dataset includes atypical images, specifically those depicting challenging scenarios, such as object occlusion within the scene, blurred images, and those with high or low illumination levels, depending on the environmental context of image acquisition.

4.1.1. Training Dataset

The training dataset comprises 28,954 samples acquired by clipping the objects from the 100 randomly selected frames belonging to the 65 videos. Each frame is aligned with the following one in the dataset since a pair of objects is required at time t-1 and time t to provide input to the network. The models trained on this type of dataset exhibit abnormal behavior: despite being trained, the neural network fails to recognize the movement of objects from one image to another, and instead, replicates the input information.

A detailed inspection of the primary dataset reveals that the input image pairs are almost indistinguishable, with a shift so slight that it is barely detectable to the human eye. If it were not for the distinct timestep and ID, it would appear that the two images depict the same scene at the exact moment in time. An example is shown in Figure 4.1: Figure 4.1a displays a frame at time t of video ID 2 from the YCB-Video dataset, while Figure 4.1b depicts the subsequent frame. Although they appear identical at first glance, these images are not the same. Figures 4.1c and 4.1d show the starting frame at time *t+10* and *t+20*, respectively, and it is evident that the objects in the scene are already moving at time *t+10*.



(a) Frame at time t

(b) Frame at time $t+1$

(c) Frame at time $t+10$

(d) Frame at time $t+20$

Figure 4.1: Comparison of frames at different time distance from the first image 4.1a of the video ID 2 of the YCB-Video dataset.

Noticing the visual similarity between successive frames, we choose to construct a histogram representing the amount of pixel scale shift between the two. With over 120,000 images in the dataset, it is unfeasible to visually examine every frame to determine if consecutive images are indistinguishable. Therefore, the most effective method to showcase this is to calculate the Euclidean distance between the centers of the objects in two frames and create a histogram of the distances. This allows for easy identification of the most frequent distance. From the dataset, we extract information on the bounding boxes of the objects $[x_{left}; y_{up}; w; h]$. Using the following formula, we derive the centers of each object in two frames

$$\begin{aligned} c_x &= x_{left} + \frac{w}{2} \\ c_y &= y_{up} + \frac{h}{2} \end{aligned} \tag{4.1}$$

from which we calculate their distance

$$d(c, \tilde{c}) = \sqrt{(\tilde{c}_x - c_x)^2 + (\tilde{c}_y - c_y)^2}. \tag{4.2}$$

After plotting the obtained data using the matplotlib library, we create a histogram. Figure 4.2 displays the distances calculated between two consecutive frames, showing that over half of the image pairs have a distance of less than 2 pixels. This result validates our earlier observation that the input images are so closely aligned that the neural network cannot determine any motion between them. Then, we create histograms using the same method for two images separated by 10 and 20 frames. This allows us to observe changes in distance. Figure 4.3 illustrates that the most common distance between images separated by 10 frames is 6 pixels. Additionally, the number of image pairs with distances less than 2 pixels significantly decreases compared to those calculated in the histogram with consecutive frames. Finally, Figure 4.4 shows the plotted distances between two frames that are 20 units apart. Similar to the previous figure, a general increase in distances is observed, resulting in larger distances occurring more frequently and the number of image pairs separated by less than 2 pixels decreasing. Noting a significant increase in distance between two images 10 frames apart, we use this configuration to generate custom dataset image pairs. Specifically, we no longer take two consecutive images from the original dataset, but rather two images with a 10-frame interval.

When dealing with pairs of images at greater distances, a new issue arises. As stated in Section 1.1.1, we need to crop the images based on the size of the bounding boxes, which a certain percentage has increased. This increase in bounding box size serves the purpose of providing context about the object's surroundings while also locating the object's movement in the next frame. Initially, the percentage was established at 10%. However, according to Table 4.1, when the distance between the two frames is increased, the likelihood of not discovering the object in the subsequent image also increases. Suppose we contemplate a pair of sequential images and only a 10% increase. In that case, there is a 99% probability of discovering the object in the following frame, of which only a small portion will partially detect the object. On the other hand, if we analyze two images 10 frames apart, the likelihood of locating the entire object significantly reduces, and the likelihood of not finding it increases.

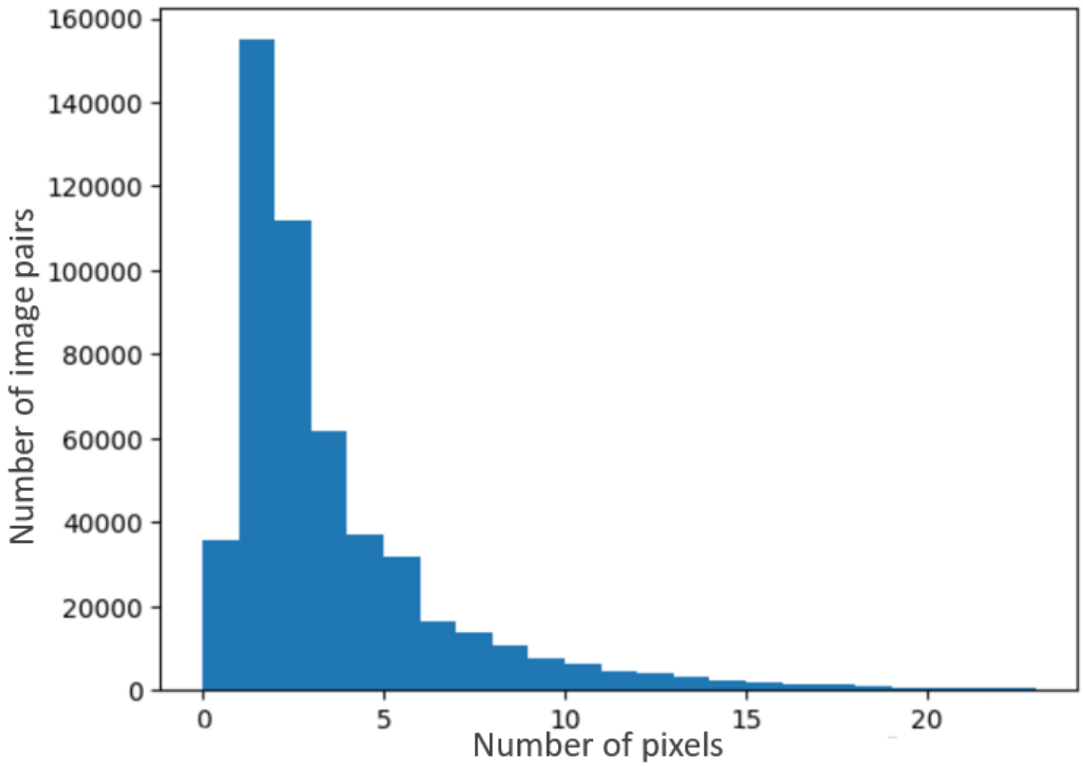


Figure 4.2: Distances between two frames at time t and time $t + 1$, measured in pixels.

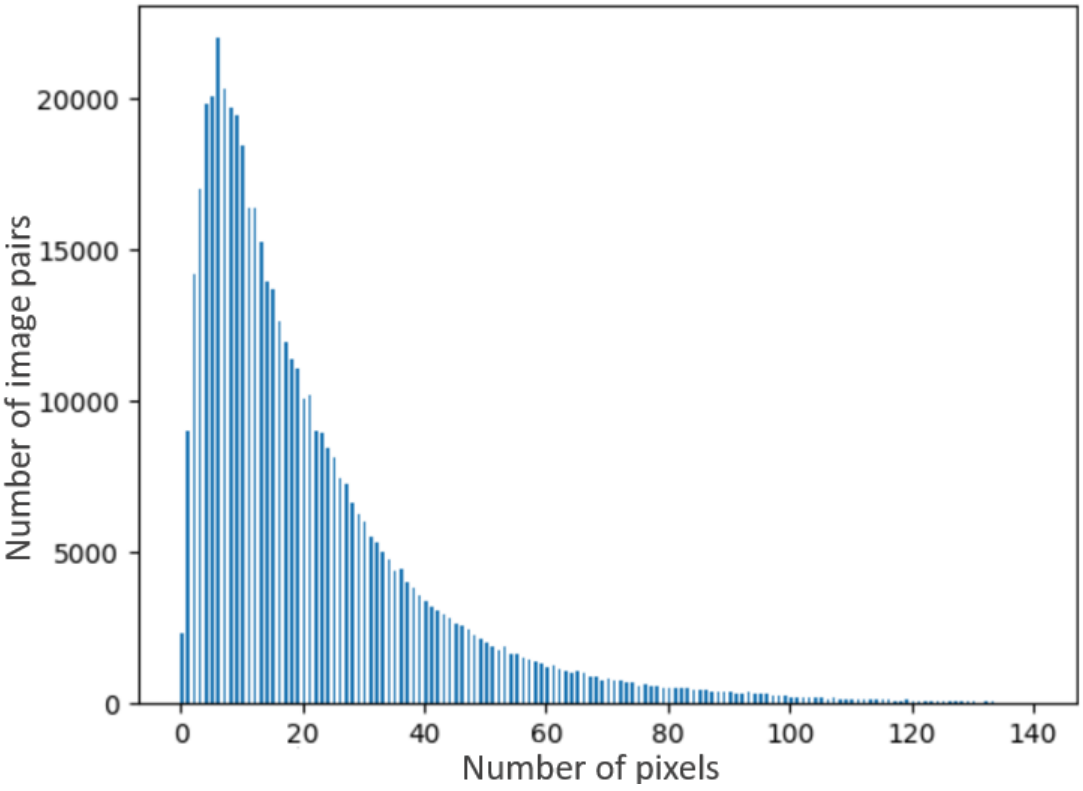


Figure 4.3: Distances between two frames at time t and time $t + 10$, measured in pixels.

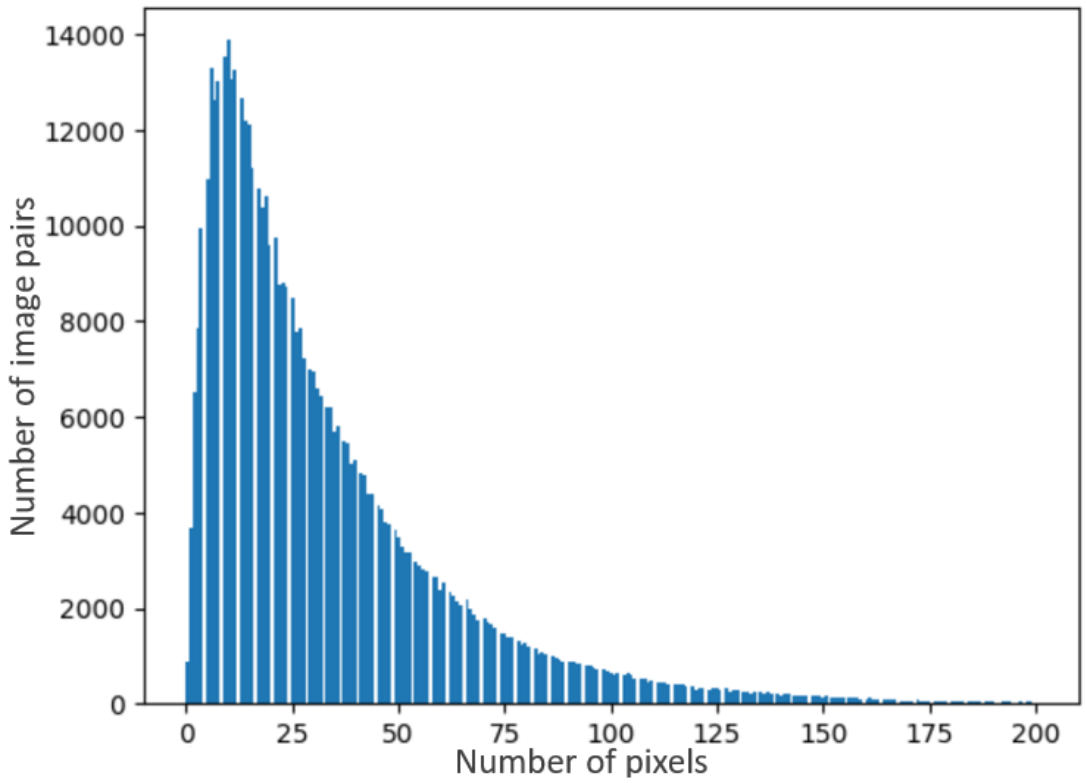


Figure 4.4: Distances between two frames at time t and time $t + 20$, measured in pixels.

This situation worsens when analyzing pictures 20 frames apart. Consequently, we adjust this hyperparameter utilizing various values, as shown in Table 4.2, to discover a setting with a higher likelihood of finding the object completely in the next image. A good solution is achieved with a 50% enlargement of the initial bounding box: the likelihood of failing to locate the object in the subsequent image 10 frames away is less than 1%. This boosts the model's chances of obtaining the associated input data accurately in training, without necessitating a major crop size increase.

	1 frame	10 frames	20 frames
Objects found (%)	97.31	48.95	32.02
Objects partially found (%)	2.62	43.51	52.17
Missed objects (%)	0.07	7.54	15.81

Table 4.1: Statistics on the number of objects detected with a bounding box increased of 10% in images taken at various frames.

	10%	30%	50%
Objects found (%)	48.95	66.44	83.93
Objects partially found (%)	43.51	29.33	15.15
Missed objects (%)	7.54	4.23	0.92

Table 4.2: Statistics on the number of objects detected with a bounding box increased by various percentages in a 10-frame interval.

Our model is specifically designed for object tracking and in practice, its input at time t is its prediction at time $t-1$. However, due to the algorithm's inability to achieve a 100% level of accuracy, errors accumulate over time, causing increasingly inaccurate predictions. To improve its resilience to this issue, we implement data augmentation into our custom dataset. The sole data augmentation we apply to the images involves translation because other transformations would not serve our ultimate goal. We initially assumed that the object would always be perfectly centered within the bounding box during the model's design phase. However, this is not the case if the model receives as input a prediction from an earlier instance. Further details on data augmentation will be presented in Section 4.4.2.

4.1.2. Validation Dataset

In the field of data science and machine learning, validation step is essential in neural network model development process. In particular, validation is a crucial step to ensure the model generalization ability, detect overfitting, tune hyper-parameters and implement early stopping. To gain insights into the model's behavior but above all to create robust and accurate models, researchers and professionals must rely on a well-constructed validation dataset.

As said, unlike training datasets that enable the learning process, validation datasets test a model's capacity for generalization. This means that the dataset chosen for this porpuse allows for the evaluation of the model's performance on unseen data, helping to determine its effectiveness and identify potential problems such as overfitting or underfitting. Therefore, comprehending the intricacies of validation datasets is crucial in refining the models and ensuring their relevance beyond the boundaries of the training data.

That being said, we have chosen to utilize four diverse datasets with varying settings to assess the algorithm's resilience and precision in different scenarios. The four validation

datasets can be classified into two primary groups based on the number of frames of distance between image pairs (i.e., 1 frame and 10 frames). These groups can then be further divided into two subgroups that depend on whether or not data augmentation is applied to the input. Therefore, the four validation datasets include the following:

1. **1-frame distance between images without data augmentation applied to the input**: In this dataset, there are pairs of images that are 1 frame apart, which represents the real setting of an object tracking algorithm. Additionally, the image pairs are cropped according to the specifications detailed in Section 2.2.1, with no modifications to the input, in order to represent the ideal conditions that one would like to have in the real application, i.e., the object placed perfectly in the center of the bounding box and aligned with the pose estimate. Given its characteristics, we refer to it as the "*Static Single Frame*" (*SSF*) validation dataset.

2. **1-frame distance between images with data augmentation applied to input**: As in the previous dataset, the images in each pair are one frame apart. However, in this case, translation is applied to both the bounding box and the pose in the input before cropping the images. The purpose of this procedure is to examine the algorithm's behavior in situations where the bounding box and pose estimate are not aligned with the center of the object in the image. Therefore, the evaluation assesses whether the model can rectify its prediction if an error occurred in the previous frame. Once again, we utilize the term "Single Frame" in this dataset's name. However, instead of "Static", we opt for "Dynamic" given the use of data augmentation, thus resulting in the name "*Dynamic Single Frame*" (*DSF*) validation dataset.

3. **10-frame distance between images without data augmentation applied to input**: In this dataset, the pair of images under consideration are 10 frames apart to evaluate the model's generalization ability over the same distance of frames used for training and to ensure that training successfully learns to estimate the required tasks. Similar to the initial dataset, no alterations are made to the input data to assess its accuracy with exact input data. For ease of reference, we refer to this dataset as the "*Static 10 Frames*" (*S-10F*) validation dataset.

4. **10-frame distance between images with data augmentation applied to input**: The latter dataset combines features from the second and third datasets. The image pairs utilized are 10 frames apart and have undergone some translation applied to the bounding box and pose, followed by image cropping. This dataset is utilized to assess the model's efficacy in rectifying the error resulting from the

previous imprecise forecasting across 10 frames. As with the other datasets, we name it the "*Dynamic 10 Frames*" (*D-10F*) validation dataset.

4.2. Evaluation metrics

When evaluating model's performance, it is crucial to take a multifaceted approach that goes beyond a single metric. To reflect the diverse goals we aim to achieve and the multifarious nature of the problem we are addressing, we have selected three distinct evaluation metrics: Intersection over Union (IoU), average distance (ADD), and average distance symmetric ($ADD\text{-}S$).

Each of the metrics provides a distinct view of our model's capabilities and comprehensively measures its effectiveness in addressing different facets of the task. The metrics are employed across all four validation datasets to systematically assess the model's performance by examining pairs of images in diverse configurations. To evaluate the effectiveness of our object tracking model in real-world scenarios, SwiftTrack is applied to three sequences of 1000 images with ground truth updates at 30 frames per second. In these three videos, we take in consideration three different objects: the *sugar box* in video 1, the *bleach cleanser* in video 2, and the *power drill* in video 3. These three objects are arranged in various settings as represented in Figure 4.5. Subsequently, the identical metrics are computed to gauge the model performance.

In the following sections, we will examine the intricacies of each metric, highlighting their unique strengths and how they offer insights into the performance of our model.

4.2.1. Intersection over Union

Intersection over Union, or IoU, is a vital evaluation metric widely implemented in the fields of computer vision and object recognition. IoU calculates the level of intersection between two areas, generally a predicted bounding box and its corresponding ground truth. This measure is crucial in tasks such as object detection and tracking, where it is used to evaluate the accuracy and precision of model predictions. It is computed by dividing the overlap of the predicted bounding box A and the ground truth bounding box B by their union. Precisely, IoU is given by

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{4.3}$$

(a) Example of a frame of video 1

(b) Example of a frame of video 2

(c) Example of a frame of video 3

Figure 4.5: Examples of frames of three videos taken in consideration to evaluate SwiftTrack performance on output propagation.

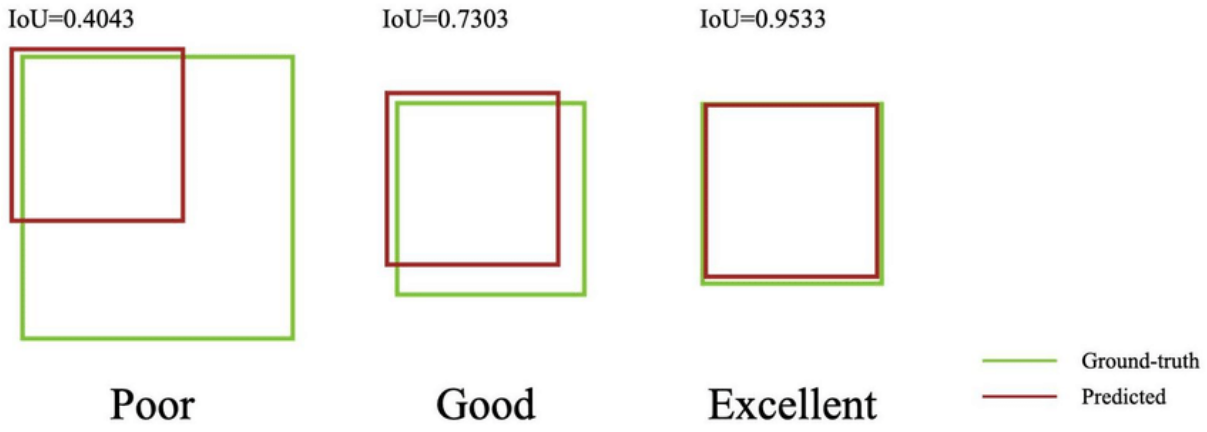


Figure 4.6: Exemples of Intersection over Union score (from [22])

As shown in Figure 4.6, a higher IoU score signifies a more precise and well-aligned prediction, as it gauges the proportion of overlap between the predicted area and the actual object.

Since the bounding box of the detected object is one of the model's key outputs, it is the most suitable metric to assess the model's effectiveness in achieving this objective.

4.2.2. 3D Object Pose Estimation Accuracy

Average Distance (*ADD*) and its counterpart, Average Distance Symmetric (*ADD-S*), are essential metrics in computer vision, particularly in the field of 3D object pose estimation and tracking. These metrics play a central role in quantifying the accuracy and precision of 3D pose estimation for objects in real-world scenes.

The ADD metric measures the average Euclidean distance between points on the model and their corresponding points on the estimated pose, providing an indication of how closely the model matches the predicted position. The formula for calculating ADD is

$$ADD = \frac{1}{m} \sum_{x_i \in M} ||(\hat{R}\mathbf{x}_i + \hat{\mathbf{t}}) - (R\mathbf{x}_i + \mathbf{t})||_2, \tag{4.4}$$

where M is the set of points in the 3D model, m is the number of points, R and $\mathbf{t}$ are the ground truth rotation and translation, $\hat{R}$ and $\hat{\mathbf{t}}$ are the predicted rotation and translation. In our case, the set M consists only of the eight points representing the edges of the 3D bounding box, since the only information available to us was the actual dimensions of the objects in the dataset. The accuracy of our ADD is not comparable to one with a larger set M, but it is enough to get an idea about the behavior of our model.

Conversely, ADD-S, a symmetric variant of ADD, evaluates the alignment between the estimated pose and the ground truth by using the closest point distance to compute the average distance between the two sets of points as in [16]:

$$ADD - S = \frac{1}{m} \sum_{x_1 \in M} \underset{x_2 \in M}{min} ||(\hat{R}\mathbf{x}_2 + \hat{\mathbf{t}}) - (R\mathbf{x}_1 + \mathbf{t})||_2, \tag{4.5}$$

where x_1 and x_2 are the points on the 3D module, R and $\mathbf{t}$ are the ground truth rotation and translation, $\hat{R}$ and $\hat{\mathbf{t}}$ are the predicted rotation and translation. Again, set M comprises the eight points of the 3D bounding box of the object under consideration, as previously described.

These two metrics are invaluable for evaluating the performance of algorithms and models responsible for estimating the position and orientation of objects. A lower ADD and ADD-S scores signify a more precise and well-aligned predictions. For this reason, we decided to use them for our model as well, since its other output is related to 3D pose estimation.

4.3. Implementation Details

We train the model on the YCB-Video dataset, as explained in Section 4.1 of the thesis, utilizing the pre-trained FlowNet-Simple to initialize the model's weights. New layer weights, including fully connected layers, are initialized randomly. The specific design of the network we train is presented in Figure 2.3. Here we detail the implementation of the network, which consists of ten convolutional layers, with a stride of 2 in six of them, utilizing the simplest form of pooling. Additionally, a LeakyReLU non-linearity is applied after every layer and the size of convolutional filters reduces towards deeper layers of neural networks. The convolutional neural network architecture utilizes a 7x7 kernel for the first layer, and 5x5 for the following two layers, then 3x3 starting from the fourth layer. The number of feature maps exhibits an increase in the deeper layers, roughly doubling after each layer with a stride of 2. Following the flattening of the features, two fully connected layers with an output dimension of 512 are present. At the end, there are four fully connected layers that are separated and have dimensions of 2, 2, 3, and 9: they output the center and dimensions of the bounding box, as well as the translation vector and rotation matrix, respectively.

The project is implemented using the PyTorch framework with the use of additional libraries such as Numpy, PyTorch Lightning, and OpenCV. The training batches contain 512 image pairs, whereas the validation batches contain 64 image pairs. NVIDIA GeForce

GTX 1080, backed by 4 CPUs, is used to train the network owing to the workload's complexity. We use the Adam optimizer, setting the learning rate at 10^{-4}. The percentage increase of the bounding box is fixed at 50% after conducting multiple experiments outlined in Section 4.1.1, and the dimensions of the input model images are set to 64 pixels in width and height. To prevent overfitting during the training process, we utilize the early stopping method with a patience of 20 epochs. This callback examines the validation dataset's loss for image pairs that are 10 frames apart and has no data augmentation, as described in Section 4.1.2. If the loss does not decrease further, the training is terminated.

4.4. Results

In this section, we thoroughly analyze the results of our experiments to gain a comprehensive understanding of the factors that affect our model's performance. During this analysis, we compare the impact of several loss functions, contrasting the outcomes achieved when four components contributes to the loss computation to those achieved with two loss components. Delving further, we examine the role of data augmentation in boosting the resilience of our model in different kinds of situations. In addition, we delve into the complexities of multi-task learning to reveal the interplay and compromises involved in training our model for simultaneous multiple objectives. The evaluation of various backbone architectures is paramount in determining their individual impact on overall performance. This allows for an assessment of the nuanced strengths and weaknesses inherent in each design.

In addition to analyzing these components, we delve into the integration of GDR-Net and our model, revealing the synergistic effects of their combination. Each subsection reveals new insights that contribute to a comprehensive understanding of the effectiveness of our approach to address the challenges at hand. They each provide a thorough account of the experiments conducted before analyzing their results through comparison.

4.4.1. Striking the Right Balance: A Deep Investigation into Loss Functions

We conduct a comparative analysis of the use of four losses versus two losses to determine the subtle effects of different loss functions on our model's overall performance. The loss function serves as a vital indicator when training machine learning models. Its essential function is to measure the extent to which the model accurately predicts in comparison to the training data. By minimizing loss, the goal is to enhance the model's capacity to

produce precise predictions. Therefore, it is crucial to have a simplified and well-organized loss function that enables the model to better learn and make precise predictions.

Section 2.2.2 highlights that the model outputs four components:

1. The coordinates of the bounding box center (c_x;c_y).
2. The dimensions of the bounding box (w and h).
3. The translation vector ($\boldsymbol{t}$).
4. The rotation matrix (R).

The most straightforward method for creating the loss function of the model with these four outputs is to add up the losses of each component. This loss which we will henceforth call *Multifaceted Loss* (L_{mul}) is illustrated by the equation:

$$L_{mul} = \alpha L_C + \beta L_{WH} + \gamma L_T + \delta L_R, \tag{4.6}$$

where L_C indicates the loss of the center of the bounding box, L_{wh} the loss of the bounding box's size, L_T the loss of the pose's translation vector, L_R the loss of the pose rotation matrix, and the coefficients α, β, γ, and δ are utilized to balance these losses. Each loss is calculated by taking the $l1$ norm between predicted $\hat{\mathbf{y}}$ and actual $\mathbf{y}$ values, as shown in the formula:

$$L_*(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{n=1}^{N} |\hat{\mathbf{y}}_n - \mathbf{y}_n|, \quad \textit{where } N \textit{ is the batch size.} \tag{4.7}$$

The varying nature of the components causes the singular loss contributions to affect the total loss in different ways: this is due to the center and dimensions of the bounding box being measured in pixels, while the translation vector is measured in millimeters and the rotation matrix is measured in radians. An example can be observed in Figure 4.7a, displaying the values of various components throughout an unbalanced training session. As depicted in the figure, the loss of the bounding box center greatly outweighs the loss of pose translation, which has minimal impact on total net loss. As a result, the training process prioritizes center loss over translation loss.

To ensure equal training of all components, the network loss must be balanced. We initially set $\alpha = 1$, ensuring that the center loss commences at an equal starting point for all other losses to be balanced according to the latter. We then add the various losses iteratively, experimenting with different coefficients for β, γ, and δ until we obtain a

combination of values that equally impact all four losses. After conducting several trials and errors, we have determined that $\beta = 4.75$, $\gamma = 1495$, and $\delta = 1200$. The changes in loss values for this configuration are illustrated in Figure 4.7b. Initially, the losses are consistently similar. However, at around epoch 40, there is a significant drop in the bounding box center loss and the translation loss, indicating the model has learned to recognize object motion.

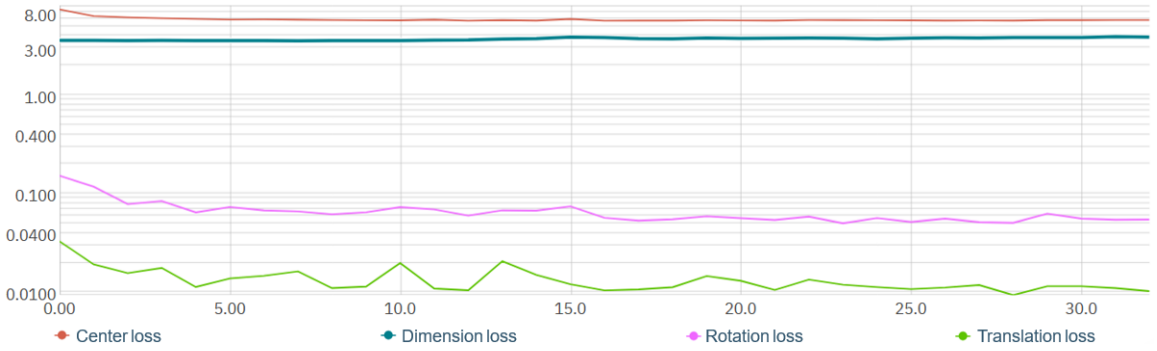
In contrast, the other two losses slightly fluctuate throughout the epochs.

Noting this trend, we decide to set the coefficients to prioritize higher losses for center and translation at the beginning, as we observed that improvements in predictions were only noticeable once these losses decreased significantly. In contrast, the bounding box size and rotation losses do not decline as much initially, so we adjust their coefficients to have a greater impact later in the training process. Our initial aim is for the network to learn the essence of motion between two images and behave appropriately. Subsequently, the focus will shift towards fine-tuning the specifics of rotation and bounding box size. The optimum coefficient values are $\beta = 1.6$, $\gamma = 900$, and $\delta = 460$. Figure 4.7c displays how the components interact under this configuration.

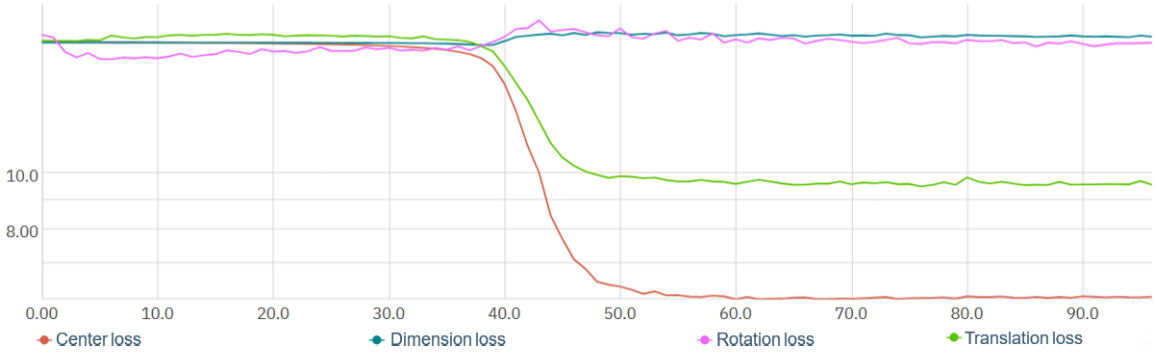
To determine the optimal coefficients for the four components of total loss, we conducted multiple experiments, but we cannot definitively conclude that these are the best values. To simplify our objective, we seek a loss representation for all network outputs with fewer components. We consolidate the center and size of the bounding box into a single vector that signifies the top-left and bottom-right corners of the bounding box. Thus, we combine two previously uncorrelated measures into a single overall measure. Instead of using the translation vector and rotation matrix separately, we opt to compute the 3D dimensions of the object's bounding box in the real world by combining them. As a result, we obtain 3D coordinates for the points in space that share the same unit of measurement. Thanks to the two unions, we are able to reduce the number of loss components from four to two. Therefore, the new formula for the network's overall loss is

$$L_{ess} = \theta L_{bbox} + (1 - \theta) L_{pose}, \tag{4.8}$$

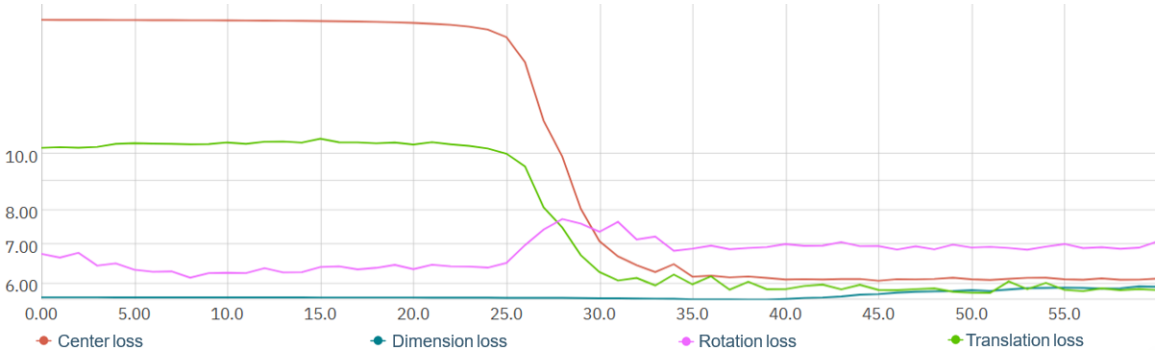
where L_{bbox} corresponds to the loss of the bounding box and L_{pose} to the loss of the pose, and from now on we will call it *Essential Loss* (L_{ess}). The L_{bbox} component is calculated utilizing the $L1$ loss, as described in Equation 4.7, however, we utilize vectors that represent the top left corner and bottom right corner of the object bounding box as predicted and ground truth values. On the other hand, the L_{pose} component is obtained by



(a) Representation of the Multifaceted Loss components during an unbalanced training session.



(b) Representation of the Multifaceted Loss components when starting with similar values.



(c) Representation of the Multifaceted Loss components when initially learning the movement and subsequently other aspects.

Figure 4.7: Comparison of the Multifaceted Loss components utilizing varying balancing coefficients.

projecting the 3D points using the estimated and ground truth pose, and then computing their $L2$ distance as in Equation 4.4.

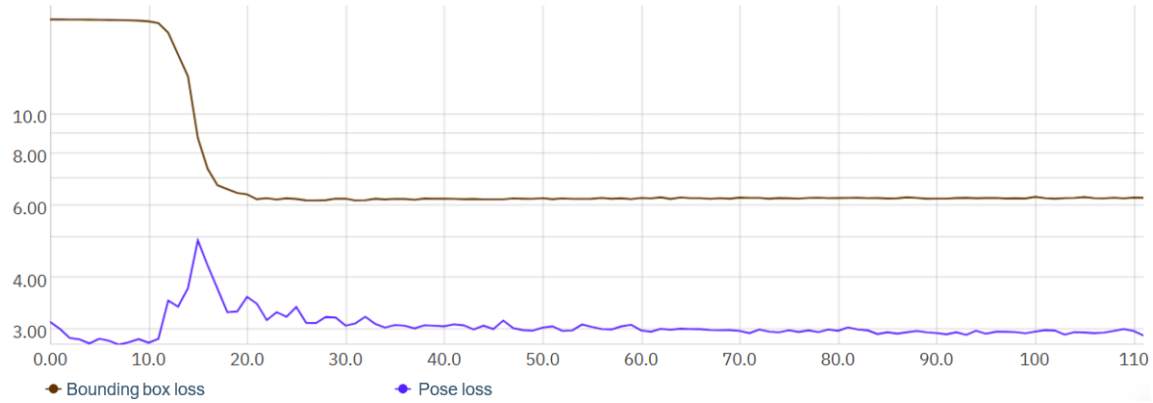
Due to the shift from four to two components for the total loss, we only need to tune one coefficient now, making the loss balancing process easier. Furthermore, the bounding box loss takes priority over the pose loss as the values for both losses significantly differ as shown in Figure 4.8a. As described above, to balance this loss, we implement two approaches:

- Setting $\theta = 0.17$ so that the two components of the network loss have equal contributions from the start, as illustrated in Figure 4.8b.
- With $\theta = 0.3$ to enable the network's initial learning of movement recognition prior to concentrating on determining the optimal pose. The trend of these losses is represented in Figure 4.8c.

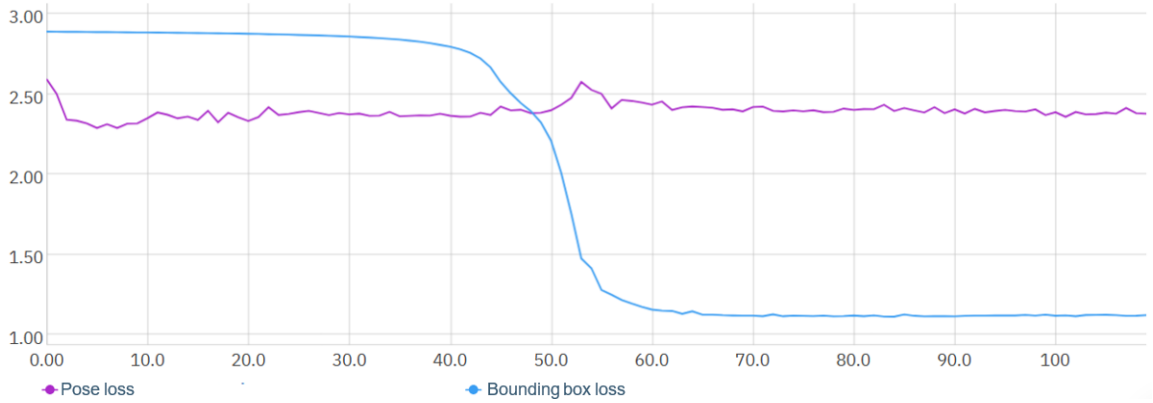
Now let examine the specifics of each experiment, analyzing the results presented in Table 4.3. The study indicates that unbalanced Multifaceted Loss performs effectively in estimating the bounding box. This is due to the high values of L_C and L_{wh}, which we have previously mentioned affect the training process to a greater extent compared to the other two factors. However, it is not successful in estimating the 3d pose. In contrast, when the L_{mul} components are balanced in the two experiments, we observe a significant improvement in the ADD metric, although there's a slight decline in the IoU. This consistency is expected as the training focuses on both estimating the bounding box and recognizing pose estimation simultaneously. While the two experiments yielded similar metric values, it is clear that they excel at opposite tasks. Specifically, the experiment that programmed the components of L_{mul} to start from a common value is superior at estimating pose compared to the experiment programmed as a scheduler. One benefit of implementing balancing with a scheduler-like behavior is the significant reduction in training time compared to other balancing method. Therefore, the preferred type of balancing can be chosen on the basis of the individual requirements.

Switching from L_{mul} to L_{ess} resulted in significant improvements in both metric quality and training speed. The table illustrates that in general all three Essential Loss experiments are more accurate than the Multifaceted Loss experiments. Additionally, the balanced coefficient experiments produce more significant results and exhibit similar behavior to the two Multifaceted Loss balanced experiments.

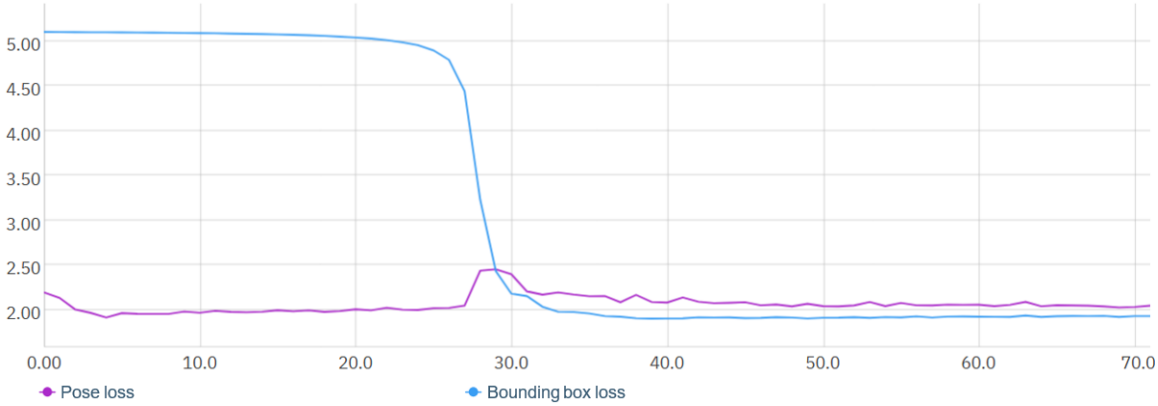
In Table 4.4, we compare the metrics calculated on three videos composed of 1000 images and the predictions are propagated between frames with ground truth updates every 30



(a) Representation of the Essential Loss components during an unbalanced training session.



(b) Representation of the Essential Loss components when starting with similar values.



(c) Representation of the Essential Loss components when initially learning the movement and subsequently other aspects.

Figure 4.8: Comparison of the Essential Loss components utilizing varying balancing coefficients.

Experiment		SSF		DSF		S-10F		D-10F	
Loss	**Weights**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**
L_{mul}	$\alpha = 1$ $\beta = 1$ $\gamma = 1$ $\delta = 1$	95.54	22.52	92.89	24.57	86.73	39.63	85.52	41.09
L_{mul}	$\alpha = 1$ $\beta = 4.75$ $\gamma = 1495$ $\delta = 1200$	94.27	4.10	92.06	11.27	86.22	8.83	84.86	14.02
L_{mul}	$\alpha = 1$ $\beta = 1.6$ $\gamma = 900$ $\delta = 460$	94.68	4.12	92.45	10.33	86.30	9.31	85.14	13.70
L_{ess}	$\theta = 0.5$	94.91	3.72	92.67	10.08	86.50	8.77	84.46	13.26
L_{ess}	$\theta = 0.17$	94.62	3.62	92.44	10.05	85.92	8.63	84.88	13.20
L_{ess}	$\theta = 0.3$	94.78	3.76	92.54	10.13	86.31	8.76	85.12	13.23

Table 4.3: Comparison between Multifaceted Loss and Essential Loss models and various balancing configurations.

frames. In each video, different objects have been taken into consideration. Although in the previous table, we observed that the IoU metric is very similar among experiments with the same balance, in this table, it is noticeable how experiments with Essential Loss are generally better than those with Multifaceted Loss, especially in video 1. Instead, if we consider the ADD metric, it is possible to notice that both in the previous table and in this table, there is a noticeable improvement with experiments using Essential Loss.

Among the experiments with Essential Loss, we don't have one experiment that stands out significantly compared to the others. The two experiments with balanced coefficients perform better in different videos: the experiment with $\theta = 0.17$ has the best results in video 2 and is the best at estimating the pose in video 1, while the experiment with $\theta = 0.3$ is better at estimating tasks in video 3 and the bounding box in video 1. The best metrics are obtained by the Essential Loss experiment with $\theta = 0.17$ in video 2: a sequence of images from the video of the experiment is shown in Figure 4.9. We can observe that the initial image provides a reliable estimate of both the 2D bounding box and the pose, benefiting from the input of the ground truth from the previous frame. However, in the subsequent images, we notice a slight deviation in the estimates from the object's ground truth, resulting in a loss of precision achieved in the first image. This discrepancy arises

Experiment		Video 1		Video 2		Video 3	
Loss	**Weights**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**
L_{mul}	$\alpha = 1$ $\beta = 1$ $\gamma = 1$ $\delta = 1$	78.50	333.63	85.80	265.12	82.69	250.85
L_{mul}	$\alpha = 1$ $\beta = 4.75$ $\gamma = 1495$ $\delta = 1200$	77.56	30.27	81.72	27.04	77.42	42.63
L_{mul}	$\alpha = 1$ $\beta = 1.6$ $\gamma = 900$ $\delta = 460$	78.90	35.97	84.45	31.94	81.84	41.13
L_{ess}	$\theta = 0.5$	82.39	29.97	78.19	26.75	73.22	49.24
L_{ess}	$\theta = 0.17$	78.90	25.70	85.38	23.24	73.85	41.56
L_{ess}	$\theta = 0.3$	80.82	31.59	82.78	24.61	80.56	39.64

Table 4.4: Comparison between Multifaceted Loss and Essential Loss models and various balancing configurations on three videos, each composed of 1000 frames and with ground truth updates every 30 frames.

from the accumulation of errors over time as predictions are propagated.

In conclusion, the shift from Multifaceted Loss to Essential Loss not only improves accuracy and performance but also accelerates the training process. This transition proves to be advantageous on multiple fronts. Furthermore, emphasizing the importance of balancing coefficients emerges as a critical factor in achieving even more refined and superior results. Indeed, we will delve into a detailed analysis of this aspect in Section 4.4.3. In summary, the combination of transitioning to Essential Loss and carefully balancing coefficients presents a robust approach for enhancing both accuracy and efficiency in the model.

4.4.2. Unveiling the Impact of Data Augmentation Technique

In this section, we delve into the intricate use of translation as a method for augmenting data as introduced in Section 4.1.1. As a crucial aspect of model training, data augmentation, and specifically translation, plays a pivotal role in enhancing the robustness and adaptability of our model against spatial variations. Our goal is to not only understand the impact of translation on performance but also to uncover its unique contributions to

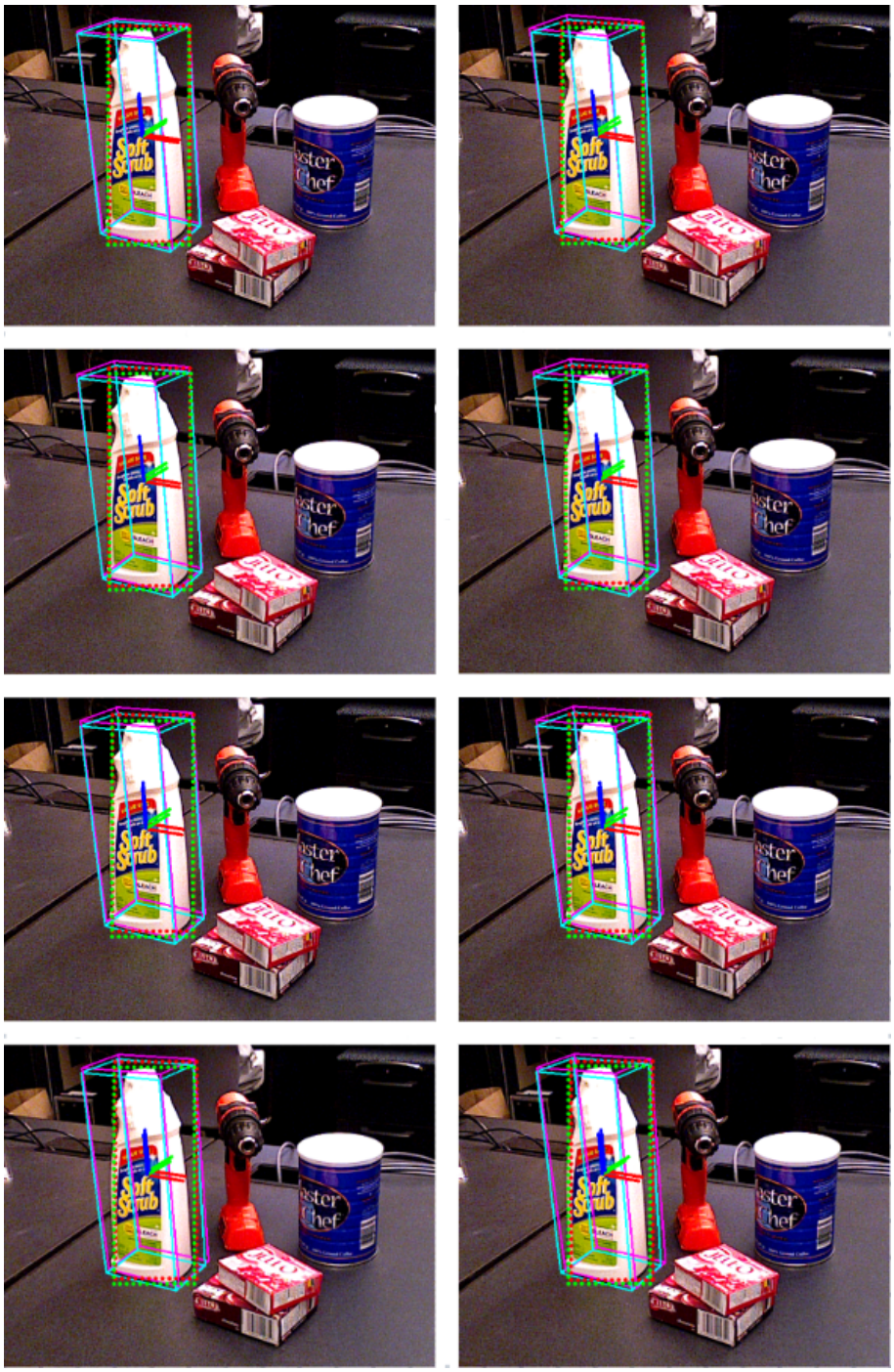


Figure 4.9: Sequence of 8 frames taken from video 2 of experiment with $\theta = 0.17$. The top-left image represents the output of the model using the ground truth from the previous frame as input. Conversely, the subsequent images (from left to right, top to bottom) illustrate estimates generated by the model using the propagated values from the previously calculated estimate. Ground truth for the 2D bounding box and object pose is denoted by green and light blue, while the model predictions are indicated by red and pink.

the model's resilience. By isolating the technique of translation, we aim to discern its specific effects on the model. Throughout this section, we will detail how translation serves as a key augmentation element, unraveling its distinctive impact and providing valuable insights into its role in improving model performance.

We opt to solely use translation as a form of data augmentation, as it mirrors the most accurate depiction of the real scenario where the prediction at instant $t-1$ deviates from the object's center point represented in the image at instant t. The process of applying this transformation is not straightforward as SwiftTrack operates on inputs with varying dimensionality: the 2D bounding box in the image plane and the 3D pose in the real world. The approach for data augmentation involves retaining the original image pairs and translating the 3D object into world coordinates. Consequently, we derive a new position in the 2D image plane, which is further utilized to select and crop the two input images. During the process, a uniform distribution is introduced to the translation vector $\mathbf{t}$ to account for any errors. Specifically, the formula used is

$$\bar{\mathbf{t}} = \mathbf{t} + U(-a, a), \tag{4.9}$$

where a indicates the maximum number of millimeters that can be shifted. After conducting various experiments, we fine-tune the hyper-parameter and achieve optimal results with $a = 0.003$, indicating a maximum displacement of approximately 6 pixels in the image plane. Subsequently, we calculate the translation $\delta\mathbf{c}$ on the image plane by utilizing the new translation vector $\bar{t}$ through the following formulas:

$$\begin{aligned} \delta c_x &= f_x(\frac{\bar{t_x}}{\bar{t_z}} - \frac{t_x}{t_z}) \\ \delta c_y &= f_y(\frac{\bar{t_y}}{\bar{t_z}} - \frac{t_y}{t_z}) \end{aligned} \tag{4.10}$$

where f_x and f_y are the focal length of the camera. Then, we calculate the new center of the bounding box $\bar{c}$ as

$$\bar{\mathbf{c}} = \mathbf{c} + \delta\mathbf{c}. \tag{4.11}$$

We apply a uniform distribution error also to the rotation R using the following formula

$$\bar{R} = R + U(-b, b), \tag{4.12}$$

where b represents the maximum number of degrees of error to be applied to the rotation matrix. Additionally, for the size $\mathbf{s}$ of the bounding box, we introduce an error using the uniform distribution with the formula

$$\bar{\mathbf{s}} = \mathbf{s} + U(-c, c), \tag{4.13}$$

where c represents the maximum number of pixels to be added as an error to the bounding box. a, b, and c are hyper-parameters we tuned through experiments to determine that the optimal setting is $a = 0.003$, $b = 2$, and $c = 5$. After introducing errors in the first part, we added a final error to the translation vector using a uniform distribution with hyper-parameter equal to 0.001 because it is uncertain if the error of the bounding box estimate matches that of the pose estimate in reality. After completing the process, the new center $\bar{c}$ and dimensions $\bar{s}$ of the bounding box values are utilized to crop a pair of images that will serve as input for the network. This same procedure is applied to the validation datasets such as Dynamic Single Frame and Dynamic 10 Frame, discussed in Section 4.1.2, where data augmentation techniques are employed.

The experiments presented in Table 4.5 are executed with consistent hyper-parameter configurations, with the sole distinction being the inclusion of data augmentation in the training dataset. Analysis of the metrics indicates that the experiment without data augmentation yielded slightly superior results compared to the experiment that employed data augmentation. Nevertheless, upon scrutinizing Table 4.6, depicting metrics in realistic scenarios, it becomes evident that data augmentation substantially enhances the accuracy of the model's predictions, making it notably more robust.

Experiment	**SSF**		**DSF**		**S-10F**		**D-10F**	
Data Augmentation	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**
No	94.91	3.72	92.67	10.08	86.50	8.77	84.46	13.26
Yes	94.05	3.84	92.41	10.12	86.11	9.30	85.25	13.70

Table 4.5: Comparison between models trained with and without the use of data augmentation in the training dataset.

Experiment	Video 1		Video 2		Video 3	
Data Augmentation	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)
No	82.39	29.97	78.19	26.75	73.22	49.24
Yes	87.50	26.21	87.12	25.79	87.41	41.14

Table 4.6: Comparison between models trained with and without the use of data augmentation in the training dataset on three videos, each composed of 1000 frames and with ground truth updates every 30 frames.

The most promising result in this comparison is observed in video 1, using the model trained with data augmentation applied to the training dataset. A sequence of 8 images from this video is shown in Figure 4.10: it is noticeable how, even in frames where the prediction from the previous frame is propagated, the estimates of the bounding box and pose are quite close to the corresponding ground truths of the object, although some error is still present.

In conclusion, we can assert that the incorporation of data augmentation into the training dataset has yielded exclusively positive effects on the model's task estimation. This enhancement has further fortified the model's ability to accurately determine the 2D bounding box and object pose, even in the presence of potential errors introduced by the propagation of predictions. The augmented dataset has proven instrumental in fostering a robust performance, contributing to the model's resilience in handling various scenarios. Consequently, the augmentation strategy emerges as a valuable tool in refining the model's capabilities and ensuring reliable predictions across diverse conditions.

4.4.3. Multi Task Learning

Multi-task learning (*MTL*) is a paradigm in machine learning that differs from the traditional approach of training models for singular objectives. Instead of solely focusing a model's proficiency in one task, MTL involves simultaneously mastering multiple, frequently related, tasks. This approach is inspired by the belief that learning numerous tasks simultaneously can enhance a model's overall performance and generalization capabilities. By integrating data from diverse tasks during training, multi-task learning creates models that exhibit a nuanced understanding of the underlying information and improved adaptability to a range of challenges. As we explore multi-task learning, it's clear that its applications span various domains, presenting a promising approach to solving complex

Figure 4.10: Sequence of 8 frames taken from video 1 of experiment with data augmentation applied to the training dataset. The top-left image represents the output of the model using the ground truth from the previous frame as input. Conversely, the subsequent images (from left to right, top to bottom) illustrate estimates generated by the model using the propagated values from the previously calculated estimate. Ground truth for the 2D bounding box and object pose is denoted by green and light blue, while the model predictions are indicated by red and pink.

problems through a unified and interconnected learning framework.

The model specializes in two tasks: estimating the 2D bounding box in the image and estimating the 3D pose in the object's world. Additionally, as discussed in Section 4.4.1, the model requires balance on the loss components to prevent learning imbalance towards one task. As discussed, we attempted to manually balance the weights, but this method presents various challenges. Model performance is highly sensitive to weight selection and this can significantly impact the results. Tuning weight hyper-parameters can be a costly and time-consuming process, often requiring several days for each trial. Consequently, it is preferable to devise a more convenient technique that can learn the optimal weights. Therefore, we attempt to apply MTL techniques to find the optimal weights and evaluate if there are substantial improvements in the estimation accuracy of our tasks compared to manual loss balancing.

Table 4.7 presents four types of multi-task learning experiments for comparison purposes:

1. **No balance**: It utilizes loss equation 4.8, in which θ is set to 0.5 to ensure equal weight for both components on the final loss. This baseline experiment allows us to assess the effectiveness of the other experiments.

2. **Random Loss Weighting** (***RLW***) [25]: It is a technique that uses randomly sampled loss weights from a distribution to train a Multi-Task Learning model. The general loss weights $\lambda = (\lambda_1, ..., \lambda_T) \in \mathbb{R}^T$ are treated as random variables and are sampled from a random distribution during each iteration. Thus, the training process employs dynamic loss weights, resembling existing loss balancing methods. However, RLW distinguishes itself by using randomized weights as opposed to the carefully designated ones used in previous works. The article contends that RWL exhibits greater potential to overcome local minima, leading to improved generalization skills. For this reason, RWL ought to be viewed as a more viable benchmark than the usual unbalanced approach.

3. **Uncertainty** [20]: It is a multitasking technique that assigns weights to multiple loss functions by taking into account the homoscedastic uncertainty of each task. Firstly, to better understand this approach, we must introduce the concept of *aleatoric uncertainty*. It refers to uncertainty regarding information that our data is unable to account for. Sequentially, we can explain that *homoscedastic* or *task-dependent* uncertainty is an aleatoric uncertainty that does not depend on the input data. It is not an output from the model, but rather a constant quantity for all input data that varies across different tasks. The level of uncertainty associated with a task reflects the relative confidence in its outcome, considering the inher-

ent uncertainty in both regression and classification tasks. It will also depend on how the task is measured and defined. This allows for the simultaneous learning of various quantities with different units or scales in both classification and regression settings.

4. **Conflict-Averse Gradient Descent** (***CAGrad***) [27]: It is a method that manipulates task gradients to alleviate the problem of conflicting gradients. When gradients of different task objectives are not well-aligned, following the average gradient direction can be detrimental to specific task performance. In practice, it minimizes the average loss function by utilizing the worst local improvement of individual tasks to regulate the algorithm, therefore promoting a consistent trajectory. The method's concept is straightforward: it searches for an update vector that maximizes the greatest local improvement of any objective within a neighborhood of the average gradient, automatically balancing different objectives and smoothly converging to an optimal point of the average loss.

All three experiments utilizing multi-task learning methods show an improvement in performance compared to the experiment with unbalanced loss coefficients, though with varying degrees of enhancement. The model employing Conflict-Averse Gradient Descent demonstrates enhancements in the estimation of the object's bounding box, but significantly worsens the pose estimation. This suggests that this method is more focused on learning the bounding box rather than the pose.

On the other hand, the algorithm using homoscedastic uncertainty has exhibited improvements in bounding box estimation across all datasets. Regarding pose estimation, there have been slight enhancements in datasets with pairs of 1 frame (SSF and DSF). This indicates that this approach balances the losses related to both bounding box and pose, striving to improve both tasks, in contrast to the previous experiment.

Experiment	**SSF**		**DSF**		**S-10F**		**D-10F**	
Balancer	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**
No Balance	94.05	3.84	92.41	10.12	86.11	9.30	85.25	13.70
RLW	94.76	3.55	92.73	9.98	86.48	9.12	85.48	13.49
Uncertainty	94.87	3.64	92.76	10.05	86.77	9.48	85.67	13.88
CAGrad	94.53	5.19	92.70	10.78	86.04	11.66	85.17	15.33

Table 4.7: Comparison between models with different Multi-Task Learning algorithms

Lastly, the experiment that improved all metrics across all datasets is the one where Random Loss Weighting was employed, demonstrating its superior effectiveness compared to a model without coefficient balancing and other multi-task learning methods

Examining the metrics obtained from experiments assessing the model's effectiveness in propagating the output from the previous frame, as outlined in Table 4.8, reveals that experiments with Uncertainty and RLW show a decrease in the IoU score. This outcome is logical since both algorithms aim to balance learning between the two tasks, unlike the experiment with unbalanced coefficients, which specializes in estimating the bounding box, as its loss predominates over the pose.

On the other hand, considering the ADD metric, distinct behaviors are observed between Uncertainty and RLW. RLW exhibits improvement in all three videos, while Uncertainty experiences a deterioration in pose estimation. This suggests that RLW is better able to balance the losses compared to the model with Uncertainty. Finally, the experiment with CAGrad demonstrates improvements in bounding box estimation in the first two videos but shows a decline in pose estimation.

In Figure 4.11, a sequence of 8 images from video 1 is depicted using the model with Random Loss Weighting as the method for balancing the components of the loss L_{ess}. It is noticeable how the predictions of the subsequent frames get progressively closer to the targets, demonstrating the model's robustness in handling the propagation of prediction errors.

Experiment	**Video 1**		**Video 2**		**Video 3**	
Balancer	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**	**IoU (%)**	**ADD (mm)**
No Balance	87.50	26.21	87.12	25.79	87.41	41.14
RLW	86.00	22.16	86.78	23.18	88.00	47.12
Uncertainty	80.07	34.17	87.09	49.15	85.85	46.99
CAGrad	89.20	34.54	88.46	43.93	86.11	52.93

Table 4.8: Comparison between models with different Multi-Task Learning algorithms on three videos, each composed of 1000 frames and with ground truth updates every 30 frames.

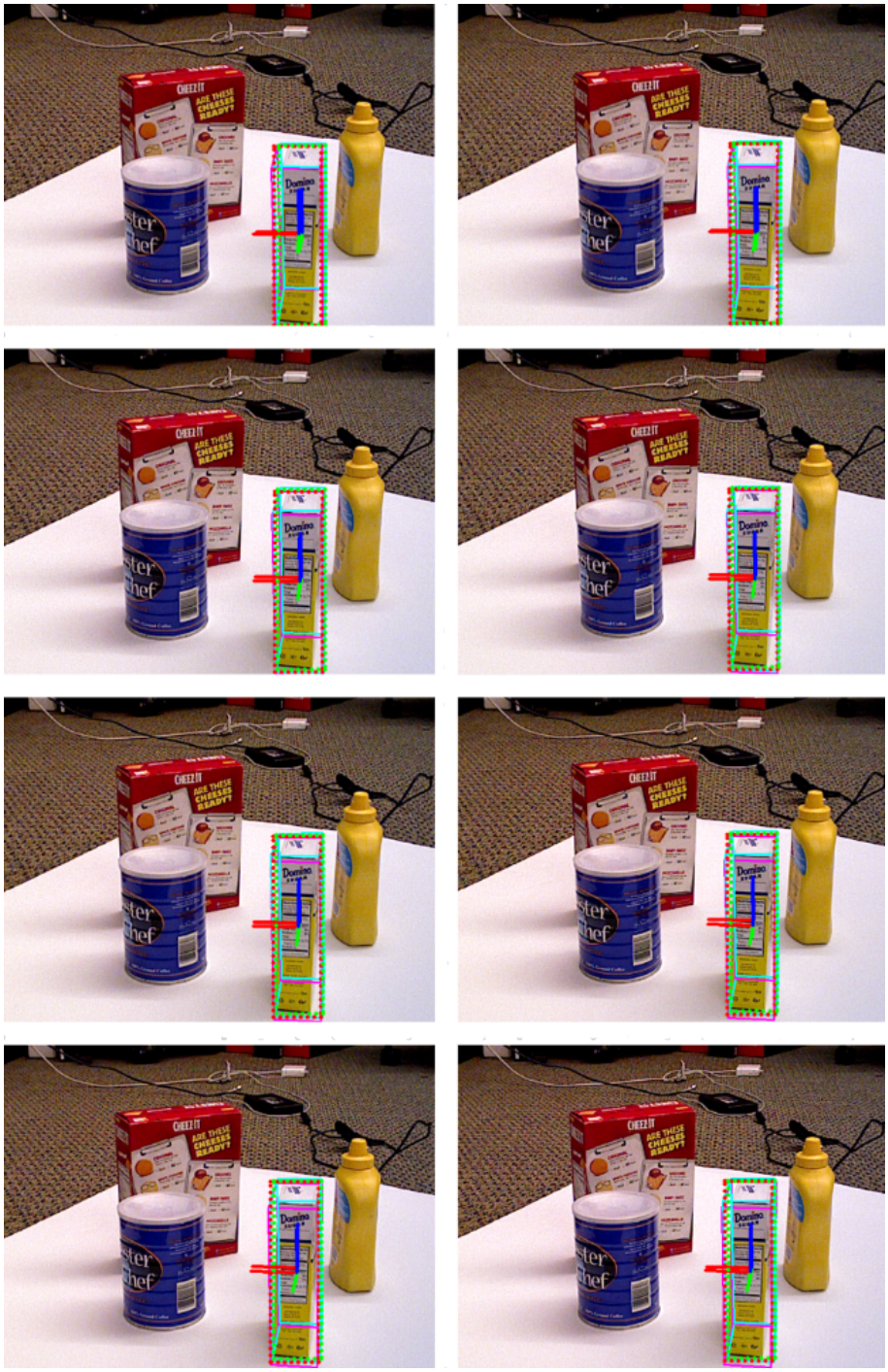


Figure 4.11: Sequence of 8 frames taken from video 1 of experiment with Random Loss Weighting method. The top-left image represents the output of the model using the ground truth from the previous frame as input. Conversely, the subsequent images (from left to right, top to bottom) illustrate estimates generated by the model using the propagated values from the previously calculated estimate. Ground truth for the 2D bounding box and object pose is denoted by green and light blue, while the model predictions are indicated by red and pink.

Therefore, the present analysis indicates that Random Loss Weighting is the most effective method, and Multi-Task Learning approaches are beneficial for multi-tasking neural networks as they enhance model performance and include an automatic means to avoid the need to search for optimal coefficients for loss network balancing.

4.4.4. Comparative Analysis of Backbone Architectures

The selection of a network backbone holds significant importance in the development of deep learning architectures, impacting both computational efficiency and the overall performance and adaptability of the model. Since the backbone acts as the foundational structure on which intricate neural networks are constructed, it becomes imperative for practitioners and researchers to grasp and compare different backbones. This comparative analysis examines different network backbones, analyzing their strengths, weaknesses, and impact on the proposed tasks. Through dissecting these intricacies, valuable insights are gained into the trade-offs between computational efficiency and model expressiveness, ultimately aiding in informed backbone selection for deep learning tasks.

The initial network utilized as the backbone of the model is **VGG16**. VGG16, also known as Visual Geometry Group 16, is a convolutional neural network architecture that was developed by the Visual Geometry Group at the University of Oxford.

First introduced in [33], VGG16 is known for its depth, comprising 16 layers, 13 of which are convolutional and 3 are fully connected, as shown in Figure 1.8. The architecture utilizes 3x3 convolutional filters with ReLU activation functions and same padding to retain spatial dimensions. It employs max-pooling layers with 2x2 filters for downsampling. VGG16 is specifically designed to handle 224x224 pixel images and serves as a popular pre-trained model for image classification tasks, especially on the ImageNet dataset. While newer architectures have surpassed VGG16's performance, it still remains a popular choice for transfer learning and serves as a foundational model in the field of computer vision. Although VGG16 has many positive qualities for transfer learning, we were unable to utilize it due to its limitation of accepting only a single RGB image as input.

Figure 4.12: VGG16 architecture (from [5])

Since our model relies on two concatenated images, this caused an incompatibility with the number of channels. Nevertheless, we decided to utilize its architectural design to evaluate its performance in object tracking and the time taken for inference.

FlowNet Simple [8] is the second network used as the backbone of the model. FlowNet Simple is a computer vision model specifically created to estimate optical flow. Optical flow is the apparent movement of objects in an image due to the relative motion between the observer and the scene. FlowNet's objective is to predict dense optical flow fields between two input frames. FlowNet Simple maintains the essential elements of convolutional neural networks for extracting features and regression layers for forecasting optical flow, as depicted in Figure 4.13. FlowNet Simple and VGG16 are different computer vision architectures designed for specific tasks and the architectural design constitutes the second discrepancy between the two models: FlowNet Simple employs a shallower network to ensure real-time performance. Unlike VGG16, transfer learning is utilized in this case due to the compatibility between the input information of the model and FlowNet Simple. Additionally, our proposed model utilizes only the initial 10 convolutional layers of FlowNet Simple, as predicting optical flow is unnecessary for our tasks.

The last backbone neural network we have tested is **Light Flow**, as presented in [45]. This model is a lighter variant of the previously mentioned FlowNet Simple. In the encoder section, the conventional 3x3 convolutions are substituted with 3x3 depthwise separable convolutions. In addition, in the decoder part, every deconvolution action is replaced with nearest-neighbor upsampling followed by depthwise separable convolution. Furthermore, the technique takes inspiration from Fully Convolutional Networks (FCN) for making multi-resolution predictions. Unlike some methods that rely solely on the most accurate optical flow prediction during inference, this study implements a technique of upscaling multi-resolution predictions to the same spatial resolution and averaging them for the final prediction. Again, only the encoder part of the general architecture was utilized in the proposed model.

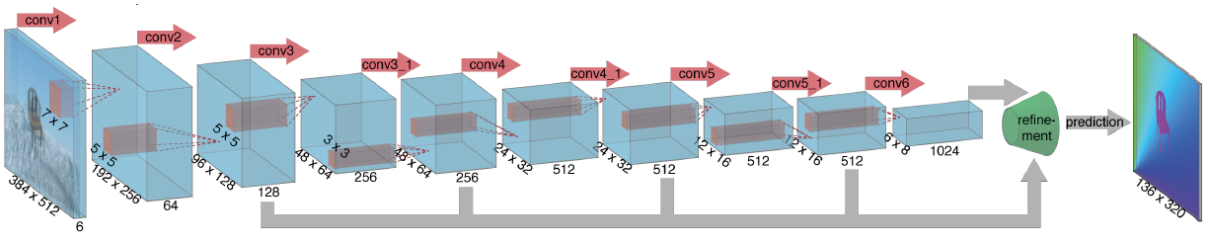


Figure 4.13: FlowNet Simple architecture (from [8])

	VGG16	FlowNet Simple	Light Flow
Number of parameters (M)	42.0	39.5	3.3
Parameters size (MB)	168.007	157.872	13.344

Table 4.9: Comparison of memory usage across various backbone models.

In these analyses, we not only evaluate the accuracy of the three architectures in predicting the 2D bounding box and 6D pose of the object, but also assess the amount of memory utilized and inference times.

As presented in Table 4.9, the Light Flow-based model has a significantly lower number of parameters, only occupying 13 MB of memory. In contrast, the other two models have a similar number of parameters, with FlowNet Simple being slightly lighter than VGG16.

All experiments in Table 4.10 are conducted with identical hyper-parameter settings to ensure comparability. VGG16 performed the worst among the three models in almost all metrics, with only slightly better values in IoU in the two datasets where data augmentation is employed. FlowNet Simple generally performed better than the other two models, but none of the three stood out as significantly better than the others.

Upon scrutinizing Table 4.11 in the context of real-world scenarios, it reaffirms the previously stated observation that none of the three architectures significantly impacts the accuracy of the estimation. Notably, Light Flow exhibits a relatively lower Intersection over Union (IoU) compared to its counterparts, namely VGG16 and FlowNet Simple, which demonstrate comparable values. Turning our attention to the Average Distance of Error (ADD), Light Flow outperforms in Video 1 and 3. However, a closer inspection of the videos reveals that these results stem from its tendency to mimic the input pose. In contrast, the remaining experiments, while displaying slightly less accuracy, exhibit a nuanced understanding of rotational aspects.

FlowNet Simple outperforms the other two models in terms of speed, with an inference time of 256 fps as demonstrated in Table 4.12. Although Light Flow ranks second, it has the least number of parameters, leading us to speculate that there may be a bottleneck in our model that's causing it to slow down. VGG16 is the slowest among all.

In contrast to earlier chapters, this section diverges from showcasing the sequence of eight images extracted from the best-performing video. Instead, it shows in Figure 4.14 frames from the Light Flow experiment to underscore the less-than-accurate nature of the predictions.

Experiment	SSF		DSF		S-10F		D-10F	
Backbone	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)
VGG16	94.07	3.30	92.04	10.51	85.32	9.34	84.47	13.86
FlowNet Simple	94.76	3.55	92.73	9.98	86.48	9.12	85.48	13.49
Light Flow	95.17	3.14	92.80	10.00	86.19	9.35	85.07	13.96

Table 4.10: Comparison between models with different backbones.

Experiment	Video 1		Video 2		Video 3	
Backbone	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)	IoU (%)	ADD (mm)
VGG16	87.28	20.93	87.23	24.10	87.41	41.14
FlowNet Simple	86.00	22.16	86.78	23.18	88.00	47.12
Light Flow	81.45	17.58	85.44	23.88	86.25	27.80

Table 4.11: Comparison between models with different backbones on three videos, each composed of 1000 frames and with ground truth updates every 30 frames.

Despite possessing the correct dimensions, it's evident that the pose is not accurately centered.

In conclusion, the ideal foundation for our model is FlowNet Simple, which surpasses all others in terms of inference speed, a crucial element for a real-time object tracking algorithm. Additionally, it provides accurate estimations for object bounding boxes and 3D poses. However, it has the drawback of utilizing slightly more memory than a Light Flow model, although not to an excessive degree. If minimum memory usage is a necessity, selecting Light Flow as the model's backbone is essential. Despite a slight decrease in efficiency and performance, the model is still capable of running in real-time, even if slower than FlowNet Simple.

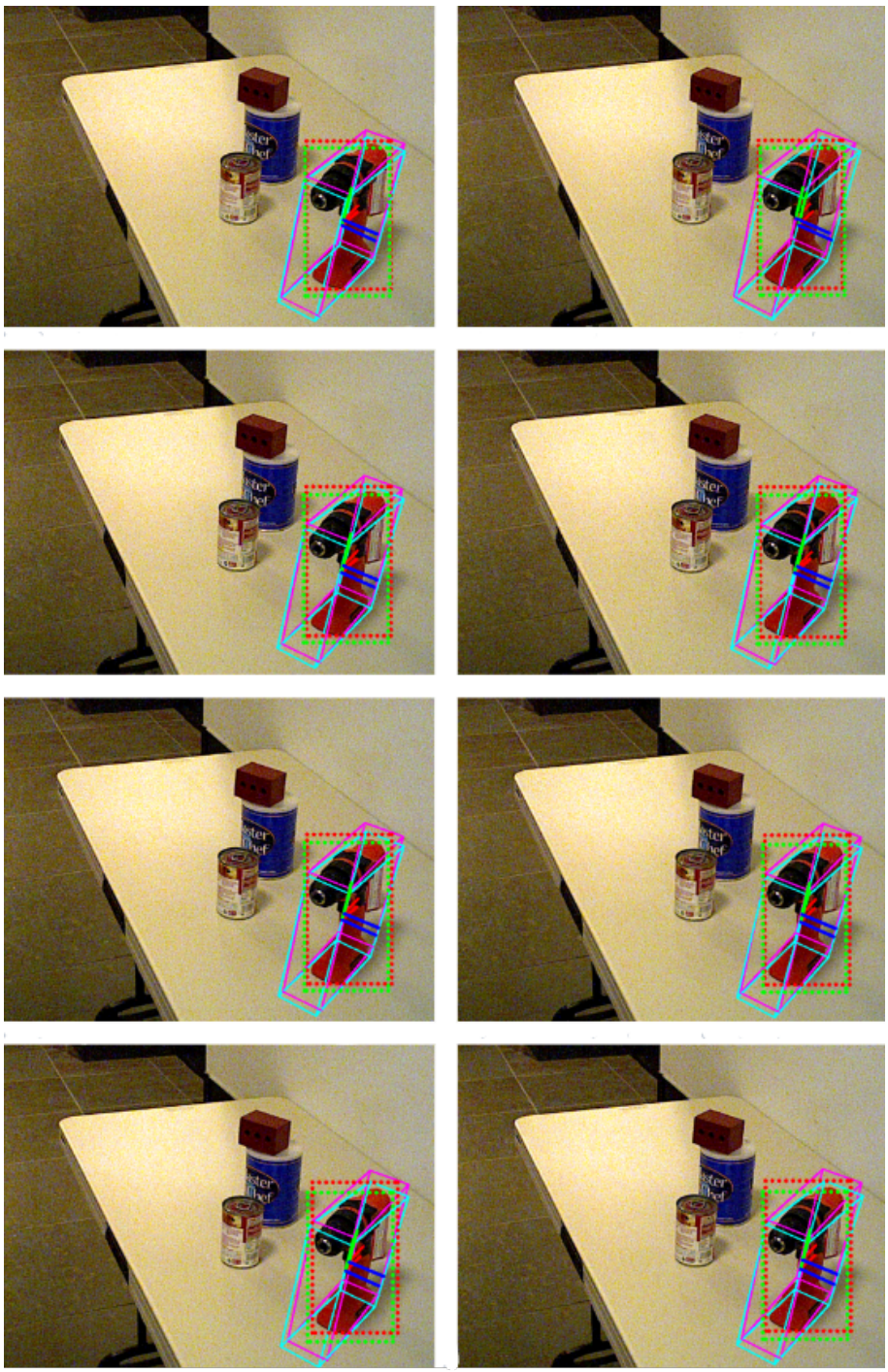


Figure 4.14: Sequence of 8 frames taken from video 3 of experiment with Light Flow architecture as backbone. The top-left image represents the output of the model using the ground truth from the previous frame as input. Conversely, the subsequent images (from left to right, top to bottom) illustrate estimates generated by the model using the propagated values from the previously calculated estimate. Ground truth for the 2D bounding box and object pose is denoted by green and light blue, while the model predictions are indicated by red and pink.

	VGG16	FlowNet Simple	Light Flow
Average Inference (*s*)	0,00519	0,00391	0,00498
Inference Variance (*s*)	0,00039	0,00040	0,00022
Average Inference (*fps*)	193	256	201

Table 4.12: Time inference of the three backbones calculated using GPU.

The majority of the experiments discussed in the preceding chapters were replicated with a change in the hyperparameter: the input size of the SwiftTrack model was adjusted from 64, as detailed in Section 4.3, to 128 pixels. This modification resulted in an enhancement of metrics across the four datasets. However, upon closer examination of metrics related to video sequences, it became apparent that the improvement is not as pronounced, particularly in addressing rotation, which still poses challenges for accurate learning.

Furthermore, as part of the data augmentation strategy, random adjustments for saturation and brightness were applied to frames during training. Despite incorporating these measures, noteworthy improvements were not observed.

4.4.5. Unleashing the Power of GDR-Net and SwiftTrack

In this conclusive section, we unveil the outcomes derived from the proposed methodology expounded in Section 4.4.5. Our findings distinctly showcase that the integration of an object tracking algorithm significantly contributes to the improvement of pose estimation method inference times, rendering it well-suited for real-time applications. The fundamental architecture of our proposal involves employing GDR-Net, the selected pose estimation model, at a lower frequency of 30 frames per second, while SwiftTrack assumes the role of predicting every frame at a higher frequency.

As illustrated in Table 4.13, SwiftTrack exhibits a noteworthy superiority in inference speed compared to GDR-Net, highlighting its exceptional efficiency in real-time task estimation. This rapid processing capability positions SwiftTrack as a primary choice for continuous use, allowing it to handle a majority of tasks seamlessly. Meanwhile, GDR-Net is strategically employed at a lower frequency to optimize pose predictions periodically. This dynamic utilization strategy not only harnesses the real-time processing prowess of SwiftTrack but also capitalizes on the specific strengths of GDR-Net, enhancing the overall accuracy of pose estimations. The flexibility in alternating between these two mod-

	GDR-Net	SwiftTrack
Average Inference (*s*)	0.01447	0.00391
Inference Variance (*s*)	0.00019	0.00040
Average Inference (*fps*)	70	256

Table 4.13: Time inference of GDR-Net and SwiftTrack calculated using NVIDIA GeForce GTX 1080.

els based on their distinct advantages contributes to an adaptive and efficient workflow, showcasing the practical versatility of the proposed framework

5 | Conclusions and Future Developments

In this concluding chapter, we bring together the key findings and insights gleaned from our exploration of 6D object pose estimation, object tracking, and the development of a novel deep learning framework.

In our exploration of embedded platforms, a key revelation emerges: the NVIDIA Jetson Nano, distinguished as the most economical, compact, and user-friendly developer board, surprisingly exhibits a performance akin to that of the CPU. This revelation implies that an inference analysis on either device can provide a reliable estimate of timings on the counterpart hardware. As anticipated, platforms endowed with superior hardware specifications demonstrate faster processing capabilities compared to their less capable counterparts. This insight underscores the importance of hardware considerations in optimizing performance across embedded platforms

Despite its straightforward structure, SwiftTrack has exhibited remarkable proficiency in accurately predicting both the bounding box and 3D pose of an object using solely RGB images. Notably robust over short intervals, the model operates at an impressive speed of 256 frames per second, ensuring real-time functionality. Additionally, SwiftTrack stands out as a lightweight model, with the flexibility to swap its backbone, FlowNet Simple, for the lighter Light Flow. While this substitution results in a slight decrease in efficiency and performance, the model remains fully capable of real-time operation. This versatility underscores SwiftTrack's adaptability to varying computational demands without compromising its core strengths.

In conclusion, our study establishes the effectiveness of the novel framework comprising GDR-Net and our object tracking algorithm, showcasing its capability to operate seamlessly in real-time scenarios. Notably flexible, this framework exhibits adaptability by accommodating various pose estimation methods utilizing RGB images. Its versatility extends to enhancing the velocity of real-time operations, positioning it as a robust solution applicable across a spectrum of pose estimation techniques. This flexibility not only

underscores the framework's efficacy but also opens avenues for widespread utilization in diverse applications requiring real-time processing.

5.1. Future Developments

As we navigate through the conclusions drawn from our experiments and analyses, we contemplate potential directions for future research and development in this ever-evolving field.

The proposed framework, as well as the object tracking model, serves as a foundational structure open to numerous enhancements for increased accuracy and efficacy. One such improvement involves the exploration of alternative datasets beyond YCB-Video. As extensively discussed in the thesis, the complexities and limitations of this dataset have posed challenges in model training. Introducing a different dataset not only addresses these issues but also broadens the algorithm's generalization capabilities, enabling it to recognize a more diverse range of objects. Further enhancements could involve experimenting with Recurrent Neural Networks to boost prediction accuracy over time, ensuring stability and reducing the frequency of pose estimation algorithm usage. Additionally, efforts can be directed towards fortifying the object tracking model in scenarios with sudden changes in light and high occlusion, enhancing its robustness. Ultimately, an additional avenue for improvement lies in refining the training of the object tracking system to adeptly identify changes in pose, with a particular focus on rotation. We have observed a notable increase in error, especially during propagation, emphasizing the need for a more nuanced approach. Despite incorporating data augmentation for rotation, the anticipated improvements did not materialize. Therefore, there is a need for a more in-depth exploration of the application of data augmentation specifically tailored to enhance the algorithm's resilience against propagated errors in rotation.

Bibliography

[1] E. Brachmann, A. Krull, F. Michel, S. Gumhold, J. Shotton, and C. Rother. Learning 6d object pose estimation using 3d object coordinates. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 536–551, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10605-2.

[2] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar. The ycb object and model set: Towards common benchmarks for manipulation research. In *2015 International Conference on Advanced Robotics (ICAR)*, pages 510–517, 2015. doi: 10.1109/ICAR.2015.7251504.

[3] R. Collins, 2020. URL `https://www.cse.psu.edu/~rtc12/CSE486/lecture12.pdf`.

[4] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[5] S. Das. CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more — medium.com. `https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5`, 2017.

[6] M. A. Dede and Y. Genc. Object aspect classification and 6dof pose estimation. *Image and Vision Computing*, 124:104495, 2022.

[7] docker. Home — docker.com. `https://www.docker.com/`, 2023.

[8] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. Van Der Smagt, D. Cremers, and T. Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[9] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24:381–395, 1981.

[10] GLuon, 2018. URL `https://cv.gluon.ai/build/examples_tracking/demo_smot.html`.

[11] D. Gordon, A. Farhadi, and D. Fox. Re^3: Re al-time recurrent regression networks for visual tracking of generic objects. *IEEE Robotics and Automation Letters*, 3(2): 788–795, 2018.

[12] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *The VLDB Journal*, 19:687–714, 2010.

[13] R. Hartley, J. Trumpf, Y. Dai, and H. Li. Rotation averaging. *Int. J. Comput. Vis.*, 103(3):267–305, July 2013.

[14] D. Held, S. Thrun, and S. Savarese. Learning to track at 100 fps with deep regression networks. In *European Conference Computer Vision (ECCV)*, 2016.

[15] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista. Exploiting the circulant structure of tracking-by-detection with kernels. In *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part IV 12*, pages 702–715. Springer, 2012.

[16] S. Hinterstoisser, S. Holzer, C. Cagniart, S. Ilic, K. Konolige, N. Navab, and V. Lepetit. Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes. In *2011 International Conference on Computer Vision*, pages 858–865, 2011. doi: 10.1109/ICCV.2011.6126326.

[17] S. Hong, T. You, S. Kwak, and B. Han. Online tracking by learning discriminative saliency map with convolutional neural network. In *International conference on machine learning*, pages 597–606. PMLR, 2015.

[18] X. Jiang, D. Li, H. Chen, Y. Zheng, R. Zhao, and L. Wu. Uni6d: A unified cnn framework without projection breakdown for 6d pose estimation, 2022.

[19] A. Kendall and R. Cipolla. Geometric loss functions for camera pose regression with deep learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5974–5983, 2017.

[20] A. Kendall, Y. Gal, and R. Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7482–7491, 2018.

[21] O. Kramer and O. Kramer. K-nearest neighbors. *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23, 2013.

[22] S.-H. Lee and H.-C. Chen. U-ssd: Improved ssd based on u-net architecture for end-to-end table detection in document images. *Applied Sciences*, 11(23):11446, 2021.

[23] F. Leeb, A. Byravan, and D. Fox. Motion-nets: 6d tracking of unknown objects in unseen environments using rgb. *arXiv preprint arXiv:1910.13942*, 2019.

[24] Y. Li, G. Wang, X. Ji, Y. Xiang, and D. Fox. Deepim: Deep iterative matching for 6d pose estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 683–698, 2018.

[25] B. Lin, F. Ye, Y. Zhang, and I. W. Tsang. Reasonable effectiveness of random weighting: A litmus test for multi-task learning. *arXiv preprint arXiv:2111.10603*, 2021.

[26] Y. Lin, J. Tremblay, S. Tyree, P. A. Vela, and S. Birchfield. Single-stage keypoint-based category-level object pose estimation from an rgb image, 2022.

[27] B. Liu, X. Liu, X. Jin, P. Stone, and Q. Liu. Conflict-averse gradient descent for multi-task learning. *Advances in Neural Information Processing Systems*, 34:18878–18890, 2021.

[28] NVIDIA, 2023. URL `https://developer.nvidia.com/embedded/jetson-modules`.

[29] Qengineering. Qengineering/jetson-nano-ubuntu-20-image: Jetson nano with ubuntu 20.04 image, 2023. URL `https://github.com/Qengineering/Jetson-Nano-Ubuntu-20-image#update-7-30-2022`.

[30] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017.

[31] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas. Frustum pointnets for 3d object detection from rgb-d data, 2018.

[32] A. Rosebrock. Opencv object tracking, Jun 2023. URL `https://pyimagesearch.com/2018/07/30/opencv-object-tracking/`.

[33] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[34] S. Song and J. Xiao. Deep sliding shapes for amodal 3d object detection in rgb-d images, 2016.

[35] R. Vock, A. Dieckmann, S. Ochmann, and R. Klein. Fast template matching and pose estimation in 3d point clouds. *Computers & Graphics*, 79:36–45, 2019.

[36] C. Wang, D. Xu, Y. Zhu, R. Martín-Martín, C. Lu, L. Fei-Fei, and S. Savarese. Densefusion: 6d object pose estimation by iterative dense fusion, 2019.

[37] G. Wang, F. Manhardt, F. Tombari, and X. Ji. Gdr-net: Geometry-guided direct regression network for monocular 6d object pose estimation, 2021.

[38] B. Wen, J. Tremblay, V. Blukis, S. Tyree, T. Müller, A. Evans, D. Fox, J. Kautz, and S. Birchfield. Bundlesdf: Neural 6-dof tracking and 3d reconstruction of unknown objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 606–617, 2023.

[39] F. Wu, Z. Wang, and Z. Hu. Cayley transformation and numerical stability of calibration equation. *International journal of computer vision*, 82:156–184, 2009.

[40] Y. Wu and K. He. Group normalization, 2018.

[41] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[42] Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes, 2018.

[43] J. Yin, W. Wang, Q. Meng, R. Yang, and J. Shen. A unified object motion and affinity model for online multi-object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6768–6777, 2020.

[44] S. Yu, D.-H. Zhai, Y. Xia, D. Li, and S. Zhao. Cattrack: Single-stage category-level 6d object pose tracking via convolution and vision transformer. *IEEE Transactions on Multimedia*, 2023.

[45] X. Zhu, J. Dai, X. Zhu, Y. Wei, and L. Yuan. Towards high performance video object detection for mobiles. *arXiv preprint arXiv:1804.05830*, 2018.

List of Figures

List of Tables

Acknowledgements

Here you might want to acknowledge someone.