ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

FLOWER POSE ESTIMATION FOR AUTONOMOUS FRUIT AND VEGETABLE CULTIVATION

Work Done In Collaboration With GrowBotHub

Gil Tinde — gil.tinde@epfl.ch

Supervised by:

Aurélien Balice-Debbas, Joachim Hugonot and Mathieu Salzmann

8th January 2021

ABSTRACT

Agricultural processes have been made to be increasingly autonomous over the past decade, and machine learning has started to play a central role in data intensive processes in order to provide valuable insights to farmers. Computer vision is being used to automate tasks such as yield prediction and crop monitoring using drones. The problem of estimating a rigid body's pose, here defined as a 3D bounding box, is not unique to agriculture but has considerable potential when it comes to vegetable growth tracking and automated harvesting. The problem to be solved in this work is how to accurately estimate the pose of a flower. The pose would provide indispensable information to an intelligent robotic arm allowing it to perceive and interact with growing fruits and vegetables. The problem of using a predicted pose to coordinate robotic actions is not discussed in this work.

The proposed solution uses a deep convolution neural network which, given an input image containing a flower, predicts a fixed number of points on the image plane that define the flower's 3D bounding box. Deep convolutional neural networks have been used extensively in image recognition because of their ability to detect highly specific features anywhere on the input images. The hypothesis behind the proposed solution is that features specific to flowers can be detected and information pertaining to a flower's orientation in 3D space can be learned such as to identify the set of points defining a flower's 3D bounding box.

Many datasets made up of images of flowers are available on the internet, however, these are primarily used for flower classification and not for pose estimation. For this reason, the dataset used to train the various networks was made from scratch using synthetic 3D models of flowers. Each individual model, also called a mesh, is rendered given a set of distinct camera views, each one producing a different image. For a given camera view, the points defining the flower's 3D bounding box, as well as these points' respective pixel mappings, are retrieved and coupled to the image. These pixel values will define the ground truth bounding box and be used in the network's loss function allowing it to learn form each input image.

A problem that often appears in rigid body pose estimation is that of accounting for an object's rotational symmetry. Flowers are, for the most part, inherently symmetrical with respect to the vertical axis Z pointing out of the flower's opening. This means that a network's predicted pose might differ greatly from the ground truth, but still qualify as a correct prediction since the two only differ in terms of the angle of rotation around the flower's vertical axis. One could argue that, for any given flower, if viewed from above (i.e., the viewer is looking directly into its opening), rotating the viewer's perspective around the flower's Z-axis by any angle $\alpha \in (0, 360)$ will produce an almost identical view of the flower. Some flowers might require α to be selected at specific intervals, but, in general, any two angles of rotation will produce unnoticeable differences unless compared side by side. This allows the problem of accounting for each flower's rotational symmetry to be solved by comparing the predicted pose to a finite number of possible rotational symmetries of the ground pose. This way, a correct yet rotated predicted pose will not be penalised nearly as much as if it was just compared to the ground truth from which it can differ greatly.

The pose estimation's performance is tested across different network structures and loss functions. Most of the models perform well on the synthetic dataset, that is their predictions are similar to how a human would define the flower's bounding box. Some of the models' predictions on real images, however, aren't quite as accurate. This might be a limitation due to the fact that synthetic data was used, and, while some high level features are identical across synthetic and real images, more fine grained features such as textures on a flower's petals are not present across the entirety of the synthetic dataset.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to Joachim Hugonot for being a great advisor. Thank you for the guidance and support provided during my research. Additionally, I want to thank all the members of the GrowBotHub project for their friendship and support. It was great pleasure to have worked together over the past few months.

Contents

CHAPTER 1

INTRODUCTION

New technologies are being increasingly incorporated into agricultural processes, and machine learning is being used to automate repetitive tasks and provide useful insight to farmers [1]. Computer vision is a subfield of artificial intelligence that seeks to develop techniques to help computers understand the content of digital images. Computer vision has many applications in agriculture [2], one of them being estimating the pose of a flower, which would allow an intelligent system, such as a robotic arm, to interact with it. GrowBotHub is a project that aims to design a fully automated, autonomous and sustainable system to grow fruits and vegetables in extreme environments. Students with various academic backgrounds are working on many different engineering challenges in order to materialize this goal.

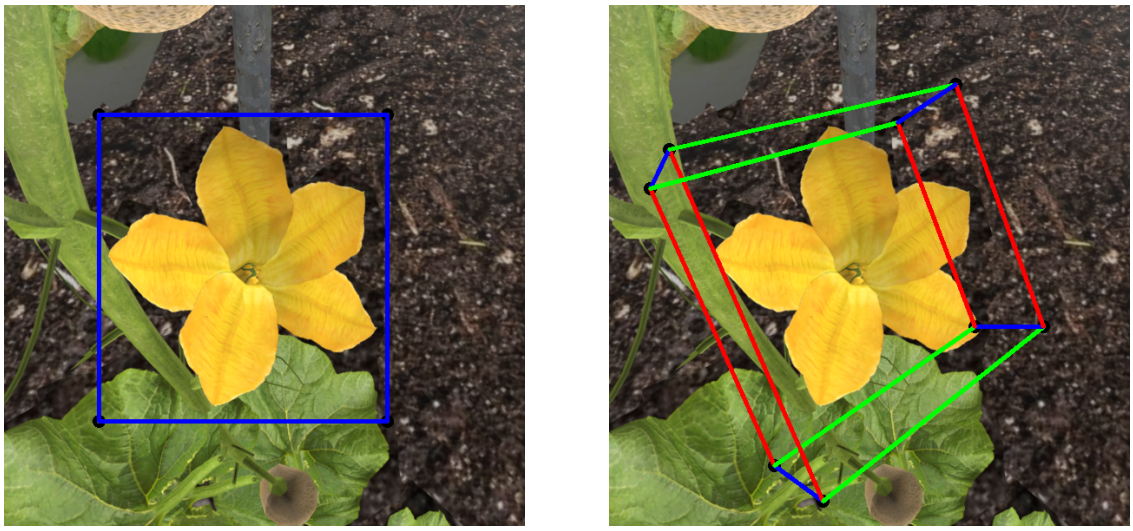
A central part of GrowBotHub's design is a robotic arm tasked with performing actions on the growing plants. In their early growth stages, however, the plants take the form of flowers and one of the actions to perform is pollination. In order to perform such an action autonomously, the robotic arm, equipped with a camera, should be able to detect the presence of a flower and estimate its pose. The idea is that perceiving its environment and estimating the flower's pose will provide enough information to the robotic arm in order to determine what movements to make next.

This work presents how a deep convolution neural network, given an input image containing a flower, can be trained to predict a fixed number of points on the image plane that define the flower's 3D bounding box. Deep convolutional neural networks have been used extensively in image recognition because of their ability to detect highly specific features anywhere on the input images [3]. The hypothesis behind the proposed solution is that features specific to flowers can be detected and information pertaining to a flower's orientation in 3D space can be learned such as to identify the set of points defining a flower's 3D bounding box. Residual networks (ResNets) are a variant of artificial neural networks that utilize skip connections in order to model certain aspects of how information is processed in the human brain. They were the architecture of choice in this work because of their computational efficiency and the state-of-the-art accuracy they can achieve [4]. Additionally, ResNets that have been trained on large, openly available datasets, such as ImageNet [5], are available to use and are already able to recognize numerous image features. Transfer learning is used in this work to take advantage of the information learned by ResNets which have been already trained to classify objects. When training a model for pose estimation, using a pre-trained network may improve the training speed and accuracy of the new model, since important image features that have already been learned can be transferred to the new task and do not have to be learned again [6]. This is referred to as "finetuning".

The network used has been pre-trained to optimize object classification and the final fully connected layer of the network, which is used to output class predictions, is replaced with a fully connected layer that outputs a fixed number of values. Given an input 600×600 RGB image containing a flower, the

output values represent the (u, v) pixel coordinates of the flower's 3D bounding box mapped onto the (U, V) image plane. A dataset consisting of images synthetic flower models was used to train the models. Because of this, after a flower has been rendered in a virtual "scene", its 3D bounding box can be easily retrieved. Each flower's 3D bounding box is defined by a set of 8 points $BBOX_{3D} = \{(x_i, y_i, z_i)\}_{i=1}^{8}$. Each point $\mathbf{p}_{3D} \in BBOX_{3D}$ lives in the 3D (X, Y, Z) "world" coordinate system. (I use the term "world" to refer to a point's position relative to the rendered scene's coordinate system and "flower" to refer to a point's location relative to the flower's own coordinate system. If neither is specified, assume that "world" is the one being referenced). Since the models are trained to output the pixel coordinates of the eight points in $BBOX_{3D}$, the number of outputs is $2 \times 8 = 16$ and can be represented by the set $BBOX_{2D} = \{(u_i, v_i)\}_{i=1}^{8}$. The problem of accounting for the flower's rotational symmetries was tackled by measuring the distance between the predicted bounding box and a finite number of the possible rotations of the ground truth bounding box.

In this work, I will refer to the set of 3D points defining a flower's 3-dimensional bounding box as its "3D bounding box" and the set of 2D pixel points defining a flower's 2-dimensional bounding box as its "2D bounding box". Any predictions and outputs from models will always be in terms of pixels, therefore they will sometimes be referred to as 3D bounding boxes in those contexts. The sets $BBOX_{2D}$ and $BBOX_{3D}$ both deal with the flower's 3D bounding box, the latter defined in terms of 3-dimensional coordinates and the former in terms of 2-dimensional pixel coordinates. The 2D bounding box is used for flower detection which is not discussed in this work.



(A) 2D bounding box used for detection.

(B) 3D bounding box used for pose estimation.

FIGURE 1.1
The difference between the 2D bounding box and the 3D bounding box. In Figure 1.1a the 2D bounding box encloses the flower in the image plane. In Figure 1.1b, on the other hand, $BBOX_{3D}$ represents the set of points in 3D space and $BBOX_{2D}$ represents how these 3D points translate to pixels on the image plane.

Chapter 2 presents the dataset used and how it was created using synthetic flower models. Chapter 3 presents the methodology regarding the model structure and how accounting for a flower's symmetries was solved using different loss functions. Chapter 4 presents the experiments done and their results. In Chapter 5 I present concluding thoughts and future work.

CHAPTER 2

DATASET

2.1 INTRODUCTION

In the early stages of the project, the dataset to be used for pose estimation was thought to be one that could be found online. However, most of the datasets available online pertain to the task of classifying different flowers. Flower detection using computer vision and deep learning methods is not uncommon [7–12], but very few groups have tackled the challenge of flower pose estimation. The proposed method in [13] classifies flower poses into one of three classes based on where the flower's center is pointing: the center points towards the center of the camera (c_1), towards the left of the camera (c_2), or towards the right of the camera (c_3). Inspired by this, the pose estimation problem was initially believed to have to be solved by transforming it into a classification problem. Unfortunately, I got no answer when I inquired about the availability of the dataset they used. I also learned that the dataset used in [12] for flower detection, which consists of real images of flowers manually labeled with 2D bounding boxes, was privately owned by a company who had no plans of releasing it. By the suggestion of my supervisor, I started looking into synthetic data in the form of 3D models of flowers, also called meshes.

The major drawback of 3D models is that they are not especially easy to work with. There is a large variety of different file types and most libraries have their own unique way of processing them. However, after a lot of trial and error, two file types, when combined with the Trimesh [14] and Pyrender [15] libraries, were found to produce satisfactory results. The models are processed by Trimesh and loaded by Pyrender into a "scene" that can be tuned to according to different criteria such as: lighting conditions, position and scale of the flower, background appearance, position of the camera that views the scene and more. In a simulated environment such as a Pyrender scene, 3D bounding boxes, depth information, instance masks and the coordinates of specific 3D points can be retrieved (e.g., the 3-dimensional coordinates of the flower's centroid). Being able to access this information was especially important during the creation of the dataset. Manually labeling the bounding boxes in thousands of images would have been laborious and far less efficient. Therefore, after overcoming the initial hurdles of setting up the model-to-image pipeline, a lot of efficiency is gained from the ability to create thousands of images in minutes.

The final dataset consists of $65{,}574$ images of flowers, each one labelled with ground truth $BBOX_{2D}$ and $BBOX_{3D}$ sets as described in Chapter 1. Additionally, the images are labelled with the coordinates of two pixels which define the flower's 2D bounding box used for flower detection. This chapter describes the important aspects of 3D models, Pyrender scenes and spatial transformations. Additionally, I cover how the actual images were generated and labelled.

2.2 MESH FILES

One thing that quickly became apparent was the lack of high quality free models of flowers. The free models that were initially used were quite easily loaded into Pyrender and very useful in creating the skeleton of the dataset creation pipeline, as well as also understanding generally how the coordinate system and transformations are set up and work in Pyrender. When it came to actually generating images, however, the free models were simply not realistic enough and looked more like toys. Furthermore, most free models used the file format .stl, which doesn't store any color or material information, so all models had to be manually colored once loaded into Pyrender which was cumbersome and added very little to how realistic the flower looked.

The .obj file format is widely used in 3D graphics and, unlike .stl files, can be accompanied by an .mtl file which specifies one or more materials, each with attributes such as transparency, ambient color and several more. The .mtl file can also specify an image to be used as a material, which is a process called UV mapping. This triplet of .obj + .mtl + images is what embodies a high quality 3D mesh. When loading an .obj file into Pyrender, it automatically looks for an associated .mtl file and, if it finds one, the loaded mesh is configured with the materials defined in the .mtl file.

The advantage of .obj files comes at a price, quite literally. There is a big market for professional grade 3D models of all kinds of imaginable objects because of the video game industry and CGI. In the early stages of the project I thought I would have to make do with free .stl models, but after exhausting a big chunk of the 3D model databases on the internet, I realized that the dataset to develop would be very poor in terms of how realistic the flower in an image looks if we couldn't get our hands on some models that were not free. After downloading a few "sample" 3D models from from a studio called Xfrog and seeing what could be achieved with them in Pyrender, I was able to convince GrowBotHub to invest in one of the bundles they offer.

The next two subsections, Section 2.2.1 and Section 2.2.2, cover details about getting synthetic models to render correctly in Pyrender and are included to serve as practical information for anyone who is attempting something similar.

2.2.1 RENDERING

3D .obj models contain a set of vertices and vertex normals which are used to define faces, implying that each face has a "front" and a "rear". When loading the professional grade models into Pyrender, the flower, or at least big parts of it, became "invisible" when viewed from certain angles. Indeed, a Pyrender rendering flag that needs to be specified in order to see the entirety of an object from all angles is "skip cull faces". In short, back-face culling means that a face that is viewed from "behind", i.e., in the opposite direction of the face normal, is not rendered. By "skipping" back-face culling, a face is rendered regardless of the viewing angle. It took some time to figure out that, for some reason, the petals of several 3D models had vertex normals that pointed away rather than towards the viewer (we imagine the viewer looks into the flower's opening along its upwards-pointing Z-axis, which I will also refer to as the "top" view, see Figure 2.2). This meant that, very frustratingly, the flower rendered in a Pyrender scene was mainly only visible from below and practically disappeared when viewed from above.

2.2.2 EDITING FILES

As stated, figuring out that setting a flag before the rendering of the scene solved the problem of the flower being invisible at certain angles. But now arised another problem: the faces that were now visible from above were technically, from the flower's perspective, being viewed from behind. This meant that they appeared darker than normal, as if the petals of the flower were a sort of transparent material viewed from behind. The solution to this problem was to use MeshLab [16], a piece of open source software used for

processing 3D meshes, and see if it wasn't possible to directly modify the 3D mesh. After watching a few tutorials about how to select specific faces of a 3D mesh and how to invert the normals of the selected faces, the flowers could be rendered correctly.

Furthermore, Trimesh recommends using the .glb file format instead of .obj because .glb is easy and fast to parse while .obj files don't have widely accepted specifications, making it tough to support. One of the problems with .obj files is arbitrary sized polygons. Indeed, all synthetic models are composed of interconnected polygons and Trimesh currently only works with models that use interconnected triangles, hence its name. This was discovered when I tried to load a mesh that used interconnected quadrilaterals. The solution was to convert the .obj file (i.e., the .obj file itself, the associated .mtl file and the images used for UV mapping) to a single .glb file. This was done using Blender [17], a free and open source software which, similarly to MeshLab, is used for 3D model creation and editing. Blender though, in my opinion, is definitely the better of the two because it has more advanced features, supports far more file formats, and is overall more stable.

2.3 SCENE SETUP

The scene in which the flower mesh is loaded is made up of a set of square meshes, each displaying its own image to simulate a hydroponic environment. There is a "base" mesh, which displays brown soil, four "wall" meshes displaying a view from a hydroponic farm and a "ceiling" mesh displaying a generic tile ceiling. Lights are added to each scene and their intensity need to be adjusted accordingly because each flower mesh has a different sensitivity to light. For example, a yellow flower could appear white if the intensity is too high while a purple flower's appearance remains unchanged in most lighting conditions. Each flower also has its own pose, i.e, spatial transformation matrix, which is described in more detail in Section 2.3.1.

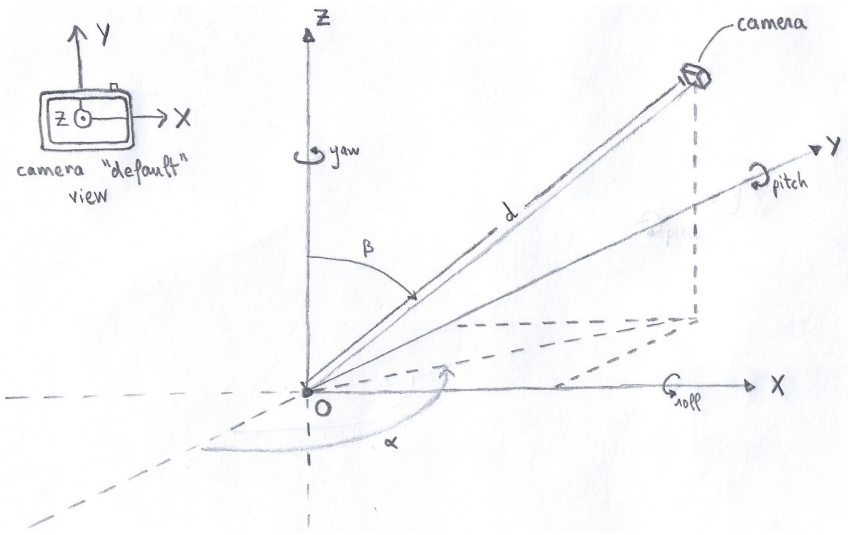


FIGURE 2.1

Diagram of a Pyrender scene and its axes. In the camera's default coordinate system the X, Y and Z axes point right, up and backwards respectively. Note that in a Pyrender scene the roll corresponds to a rotation around the X-axis, pitch corresponds to the Y-axis and yaw corresponds to the Z-axis. A set of parameters determine the camera's position. The first two are the α and β angles used to determine the camera's yaw and roll rotations respectively. A distance d to the point O determines how far the camera is from the the point it is pointed at. In this particular diagram, O is the coordinate system's center $(0, 0, 0)$, but it can be made to be any other point as well. See Section 2.3.2 for more details.

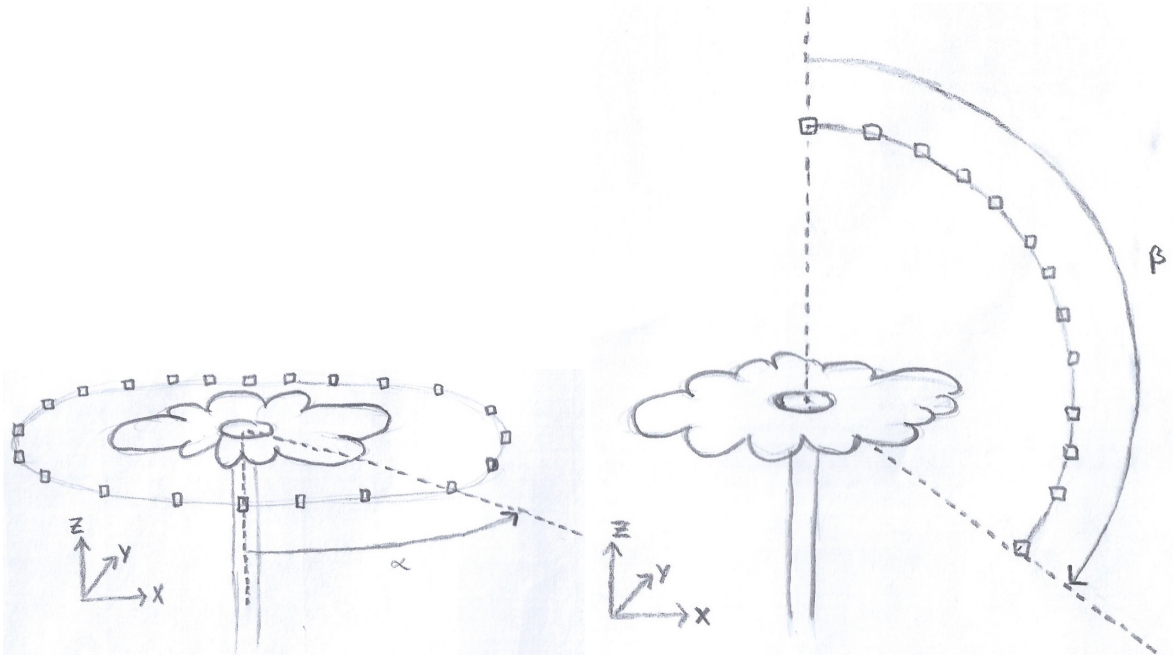
A camera is added to the scene and its position is determined according to the spherical coordinates α, β and d from Figure 2.1. For simplicity, I'll consider that the flower's center is placed at the origin $O = (0, 0, 0)$ so that we do not have to worry about specifying where the camera is pointing. In order to generate several images for a given flower model, a set containing a user-specified number of such cameras is generated, each with a distinct camera position. Consequently, the ground truth $BBOX_{2D}$ set associated to an image of a given flower will differ from another image of the same flower solely because of the change in the camera's position. The ground truth $BBOX_{3D}$ set is associated to a flower model and will be identical across all images of that flower model since the 3D mesh does not move.

The different spherical coordinates used to generate camera poses are chosen such as to place the cameras at regular intervals around the flower. The first option is for the cameras to lie on a horizontal plane and form a circle around the flower. In this case, β and d are fixed and each camera would have a distinct angle α chosen from an evenly spaced set of values in $[0, 2\pi)$. The other option is to place the flowers on a vertical plane so that they form the arc of a circle where the first camera starts with a top view of the flower and the last one ends with a side view or underneath view. I will use "side" (resp. "underneath") view to refer to a camera that points at the flower and is placed on (resp. underneath) the horizontal plane that passes through the flower's center, see Figure 2.2 for an illustration. In this second option, α and d are fixed and each camera would have a distinct angle β chosen from an evenly spaced set of values in $[0, \beta_{max}]$, where $\beta_{max} \leq \pi$. These two different ways to characterize a set of camera positions can be visualized in Figure 2.3.

Additionally, the set of poses can combine both of the aforementioned methods to form a dome of camera poses that attempts to capture all interesting viewing angles of the flower. A visualization of this dome is given in Figure 2.4. Once both the distance (d) to the flower's center and the maximum possible β value (β_{max}) have been specified, the dome is created by iterating over evenly spaced α angles and, for each one, an iteration over evenly spaced β angles takes place. In a nutshell, the camera positions start with a top view looking down at the flower's opening, move their way down by forming an arc until the β angle reaches β_{max}, then the α angle is increased a little and the process is repeated until the camera positions have moved their way all around the flower.

(A) "Top" view. (B) "Side" view. (C) "Underneath" view.

FIGURE 2.2
Different views of a flower.



(A) (B)

FIGURE 2.3
The two different classes of camera configurations where each little square represents the position of a camera pointing towards the flower's center. In Figure 2.3a the cameras are placed in a circle around the flower and each camera's position around the circle is determined by an angle α. In Figure 2.3b the cameras are placed on the arc of a vertically standing circle. Each camera's position on the arc is determined by an angle β.

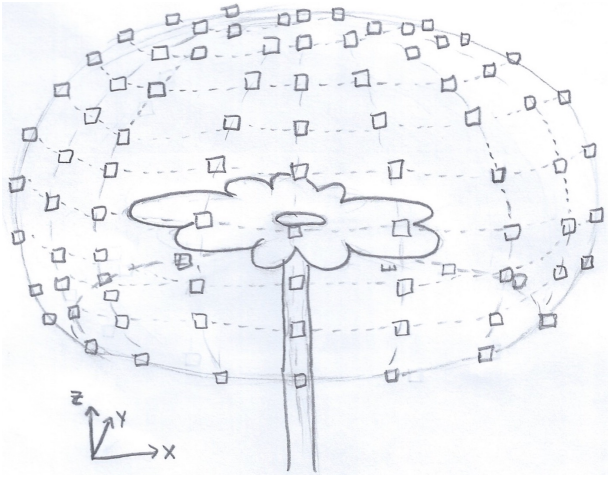


FIGURE 2.4
The two camera configurations from Figure 2.3 can be combined to form a dome of cameras, each one pointing at the flower's center but taking a different position, here represented by the little squares. In this specific visualization, α is evenly spaced in $[0, 2\pi)$ and β is evenly spaced in $[0, \frac{2\pi}{3}]$. Note that, within a given dome, two cameras' positions differ only in the α and β parameters specified to generate them. Some of the positions will have the same α or β values, but it is never the case that both parameters will match for any two camera positions. For example, the cameras placed along the centermost vertical arc in the above diagram all have the same $\alpha = 0$ angle, but have different $\beta \in [0, \frac{2\pi}{3}]$ angles.

2.3.1 SPATIAL TRANSFORMATION MATRIX

A spatial transformation matrix (STM) is used to represent the orientation and position of a set of points within the "world" coordinate system. Sub-transformations such as scale, rotation and translation are chained and combined to form a single transformation matrix. The STM is used to specify the positions of meshes in the scene such as the flower and the square "walls" as well as the position of a camera.

The STM allows one to specify the roll, yaw, and pitch rotations as well as the scale and translation transformations of a set of points within the "world" coordinate system. Let M be the STM defined by:

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the upper-left 3×3 sub-matrix represents the rotation and scaling, and the last column represents a translation. When no scaling is involved, such as moving a camera around, the STM is often written as:

$$M = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A point $\boldsymbol{v}$ is defined by a column vector:

$$\boldsymbol{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}$$

The transformation of a point $\boldsymbol{v}$ to $\boldsymbol{w}$ is obtained by applying the following matrix-vector multiplication:

$$\boldsymbol{w} = M\boldsymbol{v}$$

To reverse the previous transformation, compute M^{-1} and apply:

$$\boldsymbol{v} = M^{-1}\boldsymbol{w}$$

The 4×4 STM uses homogeneous coordinates, allowing to distinguish between points and vectors by forcing the fourth entry in a point's column to be 1 (or 0 in the case of vectors).

The three rotations (yaw, pitch and roll) together form what is called a rotation matrix. A lot of time was spent understanding these concepts because the definitions of rotation matrices and spatial transformation matrices aren't always consistent across different resources, this mainly because of specific coordinate system choices such as which axes point in which directions and whether it is a right-hand vs left-hand systems. A Pyrender scene's coordinate system is a right-handed one where the X, Y and Z coordinates point right, forward and up respectively.

The function for generating a STM, called `spatial_transform_Matrix`, takes as input: translations (t_x, t_y, t_z) along the (X, Y, Z) coordinates, scaling factors (s_x, s_y, s_z) along each axis (or one global scaling factor $s = s_x = s_y = s_z$), and the yaw, pitch and roll rotations which we represent by (α, β, γ) respectively. These specific Greek letters were chosen because they are often used in trigonometry to specify angles and it should be noted that the (α, β) pair is distinct from the α and β angles used when discussing the camera position.

The translation matrix moves a point along one or more of the three axes. The translation matrix T is defined in the following way:

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying this translation matrix to a point $\boldsymbol{v}$ results in a shift:

$$T\boldsymbol{v} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

The scaling transform changes the size of an object by expanding or contracting vertices along the three axes by three scalar values specified in the matrix. The scaling matrix S is defined in the following way:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying this scaling matrix to a point $\boldsymbol{v}$ results in a point where each component is scaled by the corresponding scaling value:

$$S\boldsymbol{v} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x s_x \\ v_y s_y \\ v_z s_z \\ 1 \end{pmatrix}$$

The rotation matrix rotates a point about one of the three coordinate axes. Therefore, the rotation matrix R is the result of multiplying three separate matrices, one for each type of rotation. Rotation matrices in 3D space are 3×3 but are 4×4 in the context of homogeneous coordinates. Additionally, 3×3 rotation matrices are orthogonal implying that $R^{-1} = R^T$. The 3×3 and 4×4 rotation matrices, denoted here by R and $R^{(4)}$ respectively, are defined in the following way:

$$R = R_z(\alpha)\, R_y(\beta)\, R_x(\gamma) = \underbrace{\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{yaw}} \underbrace{\begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}}_{\text{pitch}} \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}}_{\text{roll}}$$

$$R = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}$$

$$R^{(4)} = \left(\begin{array}{c|c} R & \mathbf{0} \\ \hline \mathbf{0} & 1 \end{array}\right)$$

Finally, the STM is obtained by applying the following chain of matrix multiplications:

$$M = TSR^{(4)}$$

2.3.2 CAMERA POSE

The previous section described how a point can be transformed using the STM, and this section will cover how to specify a camera's position, also called the camera pose. The only difference when defining a camera's position is that we pass first through a helper function, called `lookAt`, that will itself generate a STM given a set of parameters that are easier to interpret than the traditional translation, scaling and rotation parameters from the previous section.

One parameter that can be specified is the type of viewing angle the camera has of the flower, either a top view or a side view. However, if these two viewing angles are too limiting—which is often the case—two angles α and β can be passed to the function. Figure 2.1 shows what these two angles represent in the "world" coordinate system. The `lookAt` function will feed α as the yaw input and β as the roll input to `spatial_transform_Matrix`, the function generating the STM.

Furthermore, a distance needs to specified. This distance, which I call d, represents how far the camera is from the point being looked at. Using this distance with the α and β angles we end up with spherical coordinates which are used to determine the translations (t_x, t_y, t_z) to be fed to `spatial_transform_Matrix`. The translations are computed using standard spherical to cartesian conversion, taking into account the Pyrender coordinate system.

As hinted by the name of the function, three additional parameters (at_x, at_y, at_z) can be specified in order to point the camera at the point defined by those three components, instead of just pointing at the "world" origin $(0, 0, 0)$. This is particularly useful since the point at which to point the camera will often be flower's center, which is rarely close to the origin.

2.4 IMAGE GENERATION

After having placed a flower in the scene with the correct lighting conditions, images were generated by iterating over the different camera viewing angles. In order to create variation in the dataset, 3 different camera pose domes each resulting in 400 images per flower were used. It should be stated that under certain camera views the rendered flower is invisible because of obstructions such as leaves getting in the way. The generated images in which this happened were removed from the dataset by manual inspection of each image. Some images where only parts of the flowers were occluded were kept in the dataset with hopes that it would allow trained models to estimate the pose of flowers which are only partly visible. A total of 400 camera poses were used for each dome. The first dome was created by pointing each camera at the flower's center and placing it as close as possible to the flower (ensuring that the ground truth bounding boxes would largely remain within the bounds of the image plane). The second and third dome differ from each other (and from the first dome) in the distance from the flower that each camera is placed at and where it points. In the second dome, the cameras are still relatively close to the flower, but they are pointed at a point located inside the flower's 3D bounding box, the point being somewhere between the flower's centroid and one of the flower's bounding box vertices. In the third dome, the cameras are quite far from the flower and the cameras are pointed at a point located outside of the flower's 3D bounding box.

The fact that these three domes are used means that for a given flower model, there will be three sets of views of it, each of around 400 different positions. In the first set (dome 1) the flower will be centered and take up most of the image, in the second set (dome 2) the flower will be slightly smaller than in the first set and be slightly shifted off of the image plane's center, and, finally, in the third set (dome 3) the flower will be quite small with respect to the image plane's size and be shifted off of the image plane's center by a lot. The hope is that this will allow the model to learn both the features that are characteristic of a flower being viewed from a certain angle or position, and how to use these features to predict the pixel mapping of the flower's 3D bounding box. See Figure 2.5 for examples of the three dome variants.

Other changes were also made to introduce variety in the dataset such as varying the light intensities by selecting one at random from a predefined set of appropriate light intensities for a particular flower. Additionally, a different choice for the base was added towards the end of the dataset creation process: for a given camera pose a choice was made at random between showing the regular soil base and another base that tries to resemble a hydroponic farm. Finally, certain flowers looked a little lonely when rendered in the scene, so for a given camera pose, some additional green leaves were added at random (the leaves were either added or not). These variations were introduced with the hopes that the dataset could resemble real images (which certainly have a great deal of variation terms of backgrounds and lighting conditions); the resulting trained models could then hopefully accurately predict flower poses regardless of the flowers' surroundings or how well-lit they are.

(A) From dome 1. (B) From dome 2. (C) From dome 3.

(D) From dome 1. (E) From dome 2. (F) From dome 3.

FIGURE 2.5
Two flower models and three of their samples from each dome.

2.5 LABELLING

Early on it was thought that the labels to generate for a given image were the pixel coordinates of a 2D bounding box around the flower and an additional label, as stated in Section 2.1, which would be one of: the center points towards the center of the camera (c_1), towards the left of the camera (c_2), or towards the right of the camera (c_3). The former would be used for flower detection and the latter for pose estimation.

Regarding the 2D bounding box used for flower detection, its generation was first implemented by rendering the scene once with the whole flower being visible and then several other renders were made each one with an individual part of the flower ("sub-mesh") being the only one visible (i.e., stem, petals, stigma and other parts). By using the depth information of each rendered "sub-mesh" across all the scene

renderings, one can retrieve a mask of the pixels that contain the visible part of the petals. This is what's called an instance mask. From it, the minimum and maximum across the two (U, V) pixel axes can be easily retrieved to obtain the four (u, v) pixel points defining the 2D bounding box.

When it comes to pose estimation, it was thought that the three different classes (c_1, c_2, c_3) related to a flower's pose would be inferred from the different rotation angles of the camera relative to the origin of the scene's coordinates system, or wherever the flower's center was located. This is ultimately not how it was implemented and, in retrospect, would have turned out to be quite complicated because one would also have to take into account what the pose of the flower is. Indeed, even with a camera view pointing down at the flower from above, how do we know that the flower's opening is itself pointing up at the camera? It might be the case that the flower is "hanging" and tilting down a bit or rotated along one of the other axes. In such a case, labelling images is not as simple as just assigning the class c_1 to an image where the camera is placed above the flower and looking down, since there is no guarantee that the flower is "looking" directly back up at the camera. It would be cumbersome and almost boil down to manual labelling to have to handle all the possible cases because all the different poses a given flower can have would have to be taken into account, and that in addition to the camera's pose.

The other idea for pose estimation was to map specific points in the 3D (X, Y, Z) "world" coordinate system of the scene to specific pixels in the 2D (U, V) coordinate system of the image plane. Different methods were explored. The first one was to try and place a chess board in the scene (before placing the flower) to calibrate the camera and obtain a 3D to 2D pixel mapping using a set of different OpenCV [18] tools. After spending hours on implementing this functionality, it became clear that placing the chess board in the image to "calibrate" the camera was done in order to retrieve information about the camera's properties and how it captured the chess board placed in the scene. But these are things that were already available in Pyrender since it is an artificial environment. Therefore, a solution using the camera's STM and so called "projection matrix", which is directly accessible in Pyrender, was explored. A projection matrix does what it says: it projects points located in the 3D coordinate system to points located on the 2D image plane. However, this specific projection matrix does not take into account the camera's pose, so things are not as simple as just performing a matrix-vector multiplication between a point in 3D space and the projection matrix.

After some googling, I found out that a mapping from 3D "world" coordinates to 2D pixel coordinates boiled down to a chain of matrix multiplications involving the inverse of the camera's STM, the camera's projection matrix and the 3D point of interest. The problem was that there is a lot of conflicting information about which matrices to use, in which order to use them and how to implement the very last step of retrieving the pixel coordinates from the normalized vector resulting from the chain of matrix multiplications. I realized how close I actually was, actually just a mere $-$ sign was missing at one point, to getting the correct implementation, when my supervisor kindly worked out a skeleton for the solution. The mapping from a point $\boldsymbol{v}$ in the 3D "world" coordinate system to its (u, v) pixel location is defined as follows, with the projection matrix being P, the STM being M and the image width and height being w and h respectively:

$$PM^{-1}\boldsymbol{v} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$(u, v) = \left(\frac{a}{c} \cdot \frac{w}{2} + \frac{w}{2}, -\frac{b}{c} \cdot \frac{h}{2} + \frac{h}{2}\right)$$

Finally, the way to obtain the pose of a flower became to associate the set of 8 points representing the flower's 3D bounding box with the 8 pixels that correspond to the bounding box's projection onto the

image plane. As stated in Chapter 1, the ground truth 3D bounding box in terms of 3D coordinates is then $BBOX_{3D} = \{(x_i, y_i, z_i)\}_{i=1}^{8}$ and each point $\mathbf{p}_{3D} \in BBOX_{3D}$ has a 2D pixel mapping which results in the set $BBOX_{2D} = \{(u_i, v_i)\}_{i=1}^{8}$. Furthermore, the pipeline for generating the 2D bounding box also changed. Instead of rendering countless scenes for each part of the flower as previously discussed, the 2D pixel coordinates of every 3D point on the flower is computed using the above method. Then, having the pixel coordinates of every 3D point on the flower means we can retrieve the resulting minimum and maximum values across the U and V image axes, which are all that is needed to define the 2D bounding box.

CHAPTER 3

METHODOLOGY

3.1 INTRODUCTION

Rigid body pose estimation has gained a lot of attention in recent years because of its applicability in autonomous driving, robotics and augmented reality [19]. In order to determine an object's pose, certain visual features need to be detected, but defining hand-crafted feature representations is challenging and needs expert knowledge. For this reason, feature representations in images are often learned using deep convolutional networks. Several methods for performing pose estimation by relying on image feature recognition using convolutional neural networks (CNN) already exist [20–23].

Pose estimation can take different forms. A 6D pose is defined as the pose of a rigid object that has six degrees of freedom, i.e., three in translation and three in rotation. In [22] pose estimation was done by predicting such a 6D pose in 3-dimensional space. In [20, 21], however, pose estimation was done by predicting the 2D projections of the 3D control points. All three methods have quite complex architectures where the images are passed through several networks in order to optimize the task by, for example, first predicting the center of the object in an image and then applying the pose estimator to a window W centered on the object's 2D center. This work is inspired by certain aspects of the aforementioned ones, but is in no way as advanced as them. The models I trained simply take the image as input and attempt to predict the 2D pixel coordinates of a flower's bounding 3D box.

Additionally, in [23] pose estimation is performed by using synthetic models. There, an object's 3D pose in RGB images is estimated by matching keypoints to the object's CAD model, a kind of 3D synthetic model often used in engineering contexts. The method makes use of different depth renderings of a model to learn keypoints that can be transferred to the RGB domain. It would have been feasible to try and achieve something similar in this work, however, rendering depth images of the flowers was not considered when first starting to create the dataset and would have required more time to complete.

In this work, as suggested by my supervisor, the network architecture around which my methodology is based is the ResNet, which is a modified version of a classic CNN that adds skip connections. Skip connections between layers add the outputs from previous layers to the outputs of stacked layers. As shown in [4], an increase of a classic CNN's depth does not directly translate to better test results, at least not past a certain threshold. The addition of skip connections, however, widens that threshold and allows for the training of much deeper CNNs where the test results improve as the network's depth reaches up to 152 layers. Additional reasons for picking ResNets are their computational efficiency and the fact that they perform well on image classification tasks.

The models that were trained took advantage of transfer learning by modifying the final fully

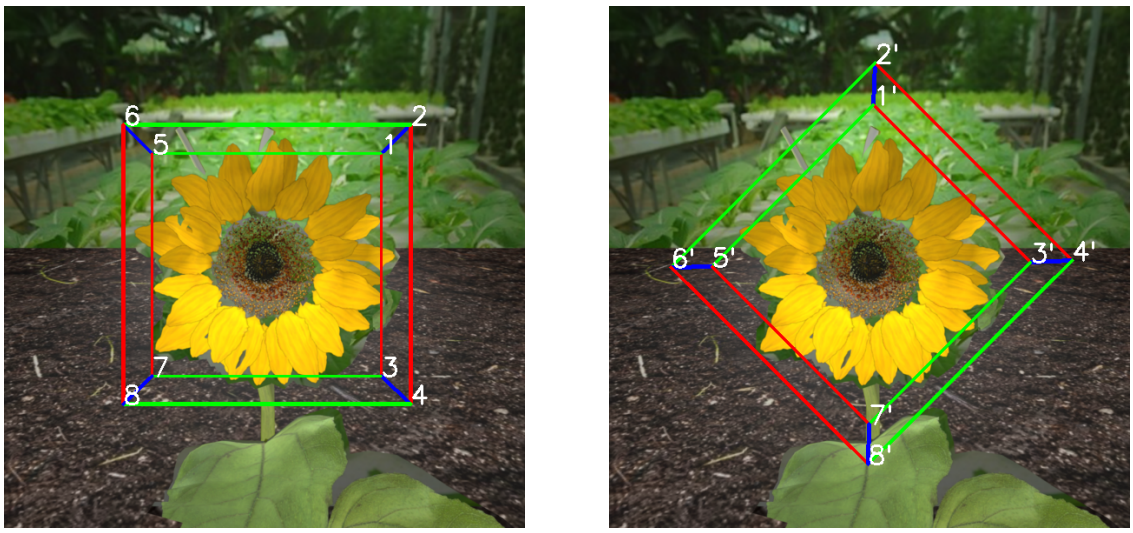
connected layer of pre-trained ResNets. Here, the word ResNets is plural because two different depths, 18 and 34, were tested. The pre-trained ResNets were initially trained to classify 1000 different objects and have a final fully connected layer with 1000 neurons to provide a predicted probability for each of the object classes. This layer was replaced with a fully connected layer that outputs the 2-dimensional pixel coordinates (one output for each U and V component) of the 3-dimensional bounding box around the flower. The default number of outputs is 16 since there are 8 points and each one has a U and V component.

The reason for performing transfer learning is that pre-trained CNNs, such as the ResNets used in this work which have been pre-trained on the openly available dataset ImageNet [5], are already able to recognize a variety of image features which reduces the training time compared to random initialization of the model weights. The hope is that image features learned for the classification task can be transferred to pose estimation, a regression task. A practical reason for choosing ResNets is that they can be directly downloaded inside PyTorch [24], the open source machine learning framework used.

3.2 LOSS FUNCTION

3.2.1 INTRODUCTION AND PLOSS

The main challenge of training a model to accurately predict the pose of a flower is figuring out how to account for the flower's rotational symmetries. In this work, all rotations are about the axis that points out of the flower's opening, which will also be referred to as the flower's vertical Z_f-axis, not to be confused with the "world" vertical Z-axis. In Figure 3.1, the ground truth 3D bounding box and one of its rotations both qualify as correct pose predictions. We can make such a claim by arguing that: (1) the four points defining the upper quadrilateral, i.e., the points labelled by the even numbers 2, 4, 6 and 8, in Figure 3.1a are coplanar to the four points defining the upper quadrilateral in Figure 3.1b, labeled by the even numbers $2'$, $4'$, $6'$ and $8'$; and (2) the four points defining the lower quadrilateral, i.e., the points labelled by the odd numbers 1, 3, 5 and 7, in Figure 3.1a are coplanar to the four points defining the lower quadrilateral in Figure 3.1b, labeled by the odd numbers $1'$, $3'$, $5'$ and $7'$. Note that the lines making up the lower quadrilateral are slightly thinner than the ones making up the upper quadrilateral, because the upper quadrilateral is in front of the flower and the lower quadrilateral is behind the flower.



(A) The ground truth bounding box.

(B) A rotation about Z_f.

FIGURE 3.1
A flower from the dataset visualized with its 3D bounding box and one of this bounding box's rotations.

The ground truth 3D bounding box $BBOX_{2D}$, which is defined in terms of 2D pixel coordinates, is stored during computations in an ordered fashion in an array: $\{1_u, 1_v, \ldots, 8_u, 8_v\}$, where the pair (i_u, i_v) refers to the pixel i from Figure 3.1a with $i = 1, \ldots, 8$. Similarly, a model will also output pixel predictions for a given image in the form of an ordered array: $\{1_u^p, 1_v^p, \ldots, 8_u^p, 8_v^p\}$, where the pair (j_u^p, j_v^p) refers to one of the predicted pixels with $j = 1, \ldots, 8$. The average squared distance, which I will call the PoseLoss (PLOSS), between a predicted bounding box $\tilde{\mathbf{b}}$ and the ground truth $\mathbf{b}$ can be then defined as:

$$\text{PLOSS}(\tilde{\mathbf{b}}, \mathbf{b}) = \frac{1}{16}\|\tilde{\mathbf{b}} - \mathbf{b}\|_2^2$$

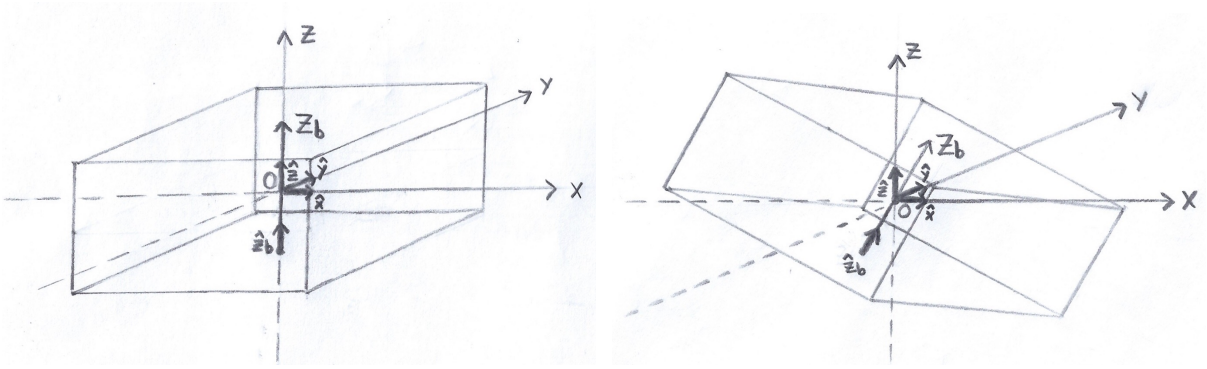
If we use the PLOSS as metric measuring the distance between the predicted and ground truth bounding box, assuming that a model predicts the rotated bounding box in Figure 3.1b and that the ground truth is Figure 3.1a, the loss computed between these two would then be very big, despite the fact that the rotated version qualifies as a correct prediction. Obviously, if we were able to rotate the ground truth bounding box by the same angle as the rotated version, we could come very close to an average squared distance equal to 0. The proposed solution to this problem draws inspiration from this fact by instead computing the distance between the predicted 3D bounding box and a finite number of the rotations of the ground truth 3D bounding box.

In the rest of this section, Z_b refers to the 3D bounding box's local vertical axis, which points from the center of its lower quadrilateral to the center of its upper quadrilateral. The "upper" and "lower" quadrilaterals use the same definition as in the beginning of this Section 3.2.1. When considering the position of the ground truth bounding box, we have that $Z_b = Z_f$. But, as we will see, the bounding box might find itself moving around so it is convenient to define its own Z_b-axis, considering that the flower does not move and its vertical axis Z_f always stays fixed in place. Also, the notation $\hat{\boldsymbol{\pi}}$ is used for the unit vector in the direction of an arbitrary Π-axis.

3.2.2 ROTATIONS IN 3D SPACE

The are different ways to rotate the ground truth bounding box about Z_f. Because each image is annotated with the $BBOX_{2D}$ *and* $BBOX_{3D}$ sets, an idea could be to rotate the points in 3D space and then project them onto the image plane. The second step of this process is simple when using OpenCV [18] tools. One simply needs to feed the $BBOX_{3D}$ and $BBOX_{2D}$ sets to a function that solves the Perspective-n-Point (PnP) problem, which is the problem of estimating the pose of a camera from n 3D-to-2D point correspondences. Using the estimated camera pose from the ground truth correspondences, the rotated 3D points can be projected onto the image plane.

The main difficulty is the first step. If a rotation matrix is applied to a set of 3D points, the points will rotate about the "world" X, Y and Z axes. This is the expected result, but the desired result is that the points rotate about the Z_f-axis, or equivalently the Z_b-axis. A way to achieve this is to apply a translation to the 3D points shifting the bounding box's centroid to the origin $(0, 0, 0)$, then apply the rotation about the world Z-axis, then finally shift the points back by applying the opposite of the first translation ensuring that the rotated bounding box's centroid has the same coordinates as before the procedure. So far, so good. However, this method only works if the unit vectors along the Z and Z_b axes, $\hat{\mathbf{z}}$ and $\hat{\mathbf{z}}_b$ respectively, both point in the same direction after the initial translation. Indeed, the shifted bounding box will rarely lay perfectly horizontally with respect to the "world" X, Y and Z coordinate system, this *only* happens when both $\hat{\mathbf{z}}$ and $\hat{\mathbf{z}}_b$ point in the same direction and I will say that the bounding box is "calibrated" in such a scenario. After shifting, the bounding box will often be tilted about the X or Y axes (or both), meaning that the Z and Z_b axes will not be aligned and I will say that the bounding box is "uncalibrated" in such a scenario. See Figure 3.2 for an illustration.

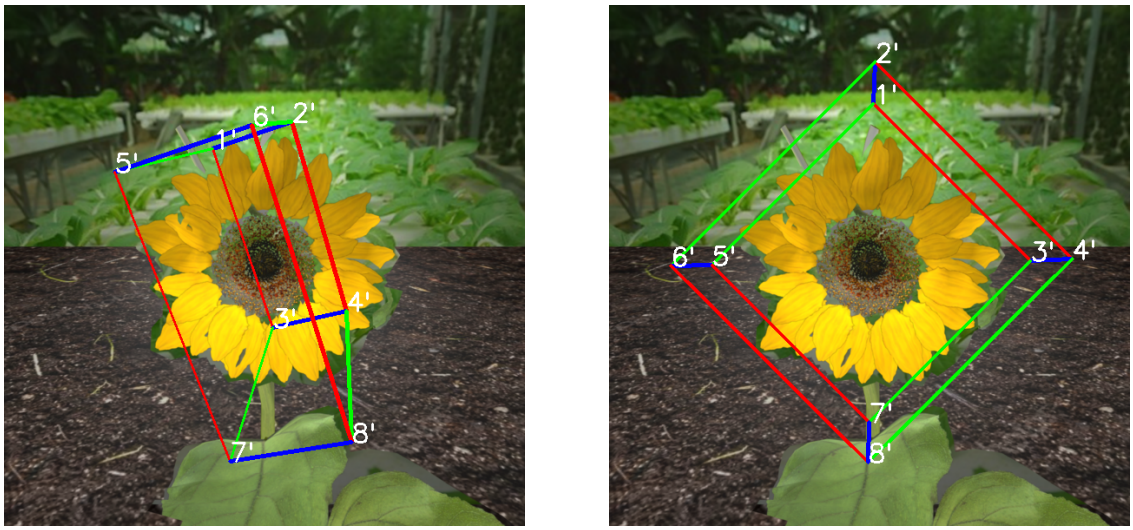


(A) The ground truth 3D bounding box, shifted to the origin and "calibrated" since $\hat{\mathbf{z}}$ and $\hat{\mathbf{z}}_b$ point in the same direction.

(B) The ground truth 3D bounding box, shifted to the origin *only*, i.e., it is "uncalibrated" since $\hat{\mathbf{z}}$ and $\hat{\mathbf{z}}_b$ point in different directions.

FIGURE 3.2
A visual comparison between a "calibrated" and "uncalibrated" 3D bounding box.

If we ignore the fact that the ground truth bounding box might be rotated about the X or Y axes, any rotation we apply about the Z-axis after translating the box to the origin will not correspond to the same rotation about the Z_b-axis, which is what we want. In Figure 3.3 the erroneous rotation obtained when not considering tilted 3D bounding boxes is compared to the desired one. In both cases the rotation is applied about the Z-axis once the bounding box has been shifted to the origin. The key point is that in Figure 3.3a, after shifting the bounding box to the origin, $\hat{\mathbf{z}}$ and $\hat{\mathbf{z}}_b$ *do not* point in the same direction before performing the rotation, but in Figure 3.3b they *do*, meaning that any rotation applied about the Z-axis will be identical to the same rotation applied about the Z_b-axis.



(A) Erroneous counter-clockwise rotation of the 3D bounding box by $\frac{\pi}{4}$. In this implementation, the rotation ends up being about the Z-axis *only*, and not the Z_b-axis.

(B) Correct counter-clockwise rotation of the 3D bounding box by $\frac{\pi}{4}$. In this implementation, the rotation is about both the Z and Z_b axes.

FIGURE 3.3
A comparison of the rotation of a flower's 3D bounding box, with Figure 3.3a being the erroneous implementation, and Figure 3.3b being the correct version.

To solve this problem, we need to apply an additional rotation to the 3D bounding box, *before* attempting any rotation about the Z-axis. After shifting to the origin, this separate rotation R will align the Z_b-axis with the Z-axis, i.e., "calibrating" the bounding box. After the calibration step, we can apply the desired rotation, then "uncalibrate" the bounding box by applying the inverse rotation R^{-1}, ensuring that $\hat{\mathbf{z}}_b$ points in the same direction as it originally did. The final step is to simply shift the bounding box back to the flower. At the end of the procedure, $\hat{\mathbf{z}}_b$ and $\hat{\mathbf{z}}_f$ will point in the same directions just as they initially did. The rotation R, which takes the form of a 3×3 matrix as discussed in Section 2.3.1, can be obtained in the following way: given a pair of normalized vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, finding the rotation matrix R that will rotate $\boldsymbol{a}$ onto $\boldsymbol{b}$ can be achieved by computing:

$$\begin{aligned} \boldsymbol{v} &= \boldsymbol{a} \times \boldsymbol{b} \\ s &= \|v\| \\ c &= \boldsymbol{a} \cdot \boldsymbol{b} \end{aligned}$$

Then, the rotation matrix R is given by:

$$R = I_3 + [\boldsymbol{v}]_\times + [\boldsymbol{v}]_\times^2 \frac{1-c}{s^2}$$

where $[\boldsymbol{v}]_\times$ is the skew-symmetric cross-product matrix of $\boldsymbol{v}$:

$$[\boldsymbol{v}]_\times \stackrel{\text{def}}{=} \begin{pmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{pmatrix}$$

In our case, $\boldsymbol{a}$ will be $\hat{\mathbf{z}}_b$, the unit vector in the direction of the Z_b-axis, and $\boldsymbol{b}$ will be $\hat{\mathbf{z}}$, the unit vector in the direction of the Z-axis. In order to "calibrate" the bounding box after it has been shifted to the origin, we apply a matrix multiplication between R and the matrix form of the eight 3D points constituting the flower's bounding box, B, to obtain the calibrated version C:

$$C = RB$$

where B is the matrix form of $BBOX_{3D}$:

$$B = \begin{pmatrix} 1_x & 2_x & 3_x & 4_x & 5_x & 6_x & 7_x & 8_x \\ 1_y & 2_y & 3_y & 4_y & 5_y & 6_y & 7_y & 8_y \\ 1_z & 2_z & 3_z & 4_z & 5_z & 6_z & 7_z & 8_z \end{pmatrix}$$

In order to "uncalibrate" C to obtain B simply apply:

$$B = R^{-1}C = R^T C$$

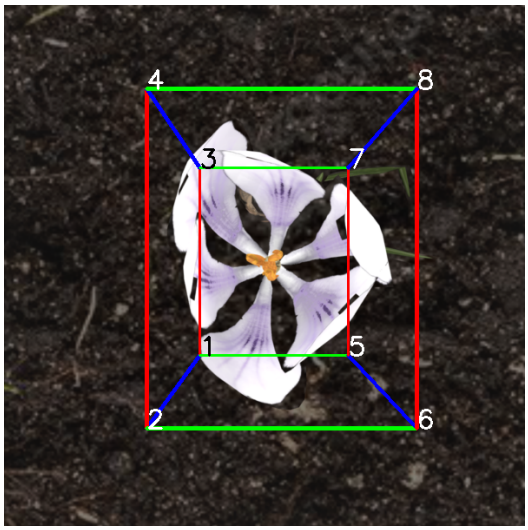
Indeed, $R^{-1} = R^T$ as was seen in Section 2.3.1.

In summary, the steps are:

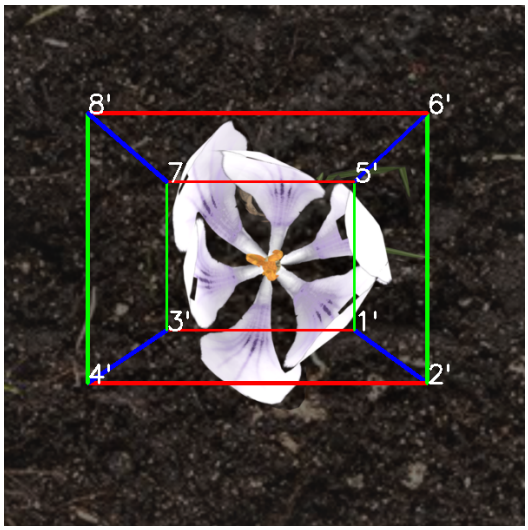
1. Compute the centroid $c = (c_x, c_y, c_z)$ of the bounding box defined by the points in $BBOX_{3D}$.
2. Subtract the centroid from every point in $BBOX_{3D}$ component-wise, i.e., for a point $\mathbf{p} \in BBOX_{3D}$, its new coordinates will be $(p_x - c_x, p_y - c_y, p_z - c_z)$.
3. Obtain C, the calibrated version of the bounding box by computing the rotation matrix R and performing $C = RB$, where B is the matrix form of the shifted $BBOX_{3D}$.
4. Apply to C a rotation about the Z-axis of an angle $\gamma \in (0, 2\pi)$, resulting in C'.

5. Uncalibrate the rotated bounding box by applying $B' = R^T C'$.
6. With B' in set form as $BBOX'_{3D}$, add the centroid c computed in the first step to every point in $BBOX'_{3D}$ component-wise, ensuring that the centroids of the original and rotated bounding boxes have the same coordinates in 3D space.

It seems as though we have achieved what we wanted. By applying the above steps with evenly spaced $\gamma \in (0, 2\pi)$ angles we can obtain a finite set of the possible rotations of a flower's bounding box in three dimensions. As discussed earlier, each 3D point of the rotated bounding boxes can be mapped onto the image plane by making use of the PnP problem's solution available in OpenCV. However, simply rotating the bounding box is not always correct. Take the example given in Figure 3.4. There, the bounding box's upper and lower quadrilaterals are no longer squares as we have previously seen, but instead rectangles. Since a bounding box is defined as a set of points enclosing an object, we can conclude that the one in Figure 3.4b does not fully satisfy this definition, as the face defined by the points $\{1', 2', 3', 4'\}$ does not reach as far down in the image as the face defined by the points $\{1, 2, 5, 6\}$ does in Figure 3.4a, the former allowing parts of the flower to protrude out of the bounding box.



(A) The flower with its ground truth 3D bounding box



(B) The flower with the ground truth 3D bounding box having been rotated by $\frac{\pi}{2}$ radians counter-clockwise.

FIGURE 3.4
A comparison of a flower's 3D bounding box, with Figure 3.4a being the ground truth, and Figure 3.4b being the rotated version.

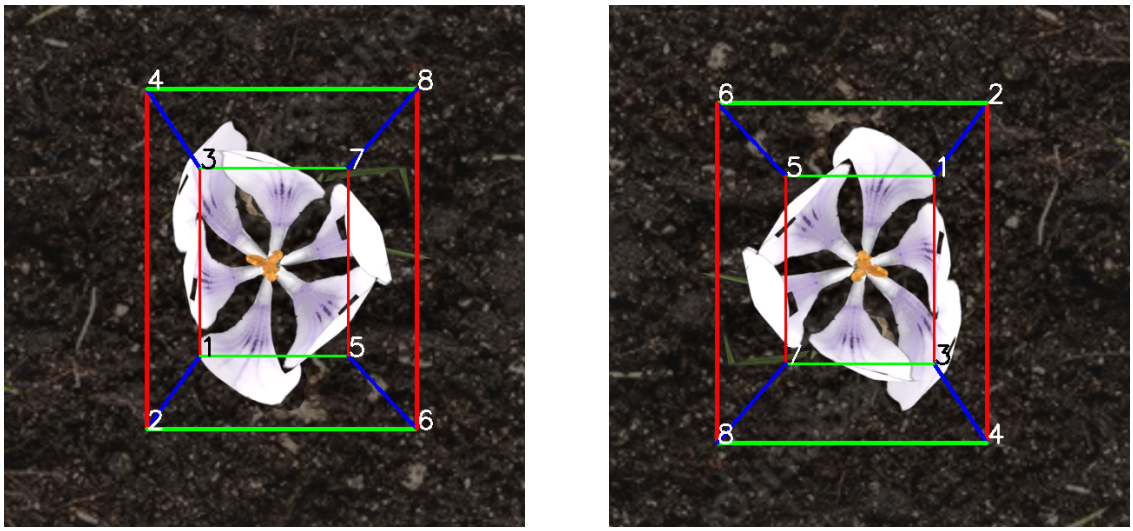
One could argue that the flower model in Figure 3.4b is not symmetrical about Z_f, and therefore the problem of accounting for its rotational symmetries disappears, meaning that computing the average squared distance between the predicted and ground truth bounding box could be sufficient to train an accurate model, i.e., using the already defined PLOSS. As Chapter 4 experimentally shows, this is not the case, but here I will attempt to give an intuitive explanation of what the problem is and a possible solution.

The points in the ground truth $BBOX_{3D}$ set of a flower model remain fixed in place since, as we have said, the flower mesh itself does not move and all $BBOX_{3D}$ sets are identical for all images of the same flower model. The differences in the $BBOX_{2D}$ sets for the same flower model arise from the changes in the camera poses. In Figure 3.5, two images of the same flower model are displayed. It is the same flower model as the one used in Figure 3.4. I will denote the $BBOX_{2D}$ sets of the images in Figure 3.5a and Figure 3.5b as $BBOX^A_{2D}$ and $BBOX^B_{2D}$ respectively. During computations the sets are

stored in arrays where the pixels have a specific order:

$$BBOX^A_{2D} = \{1^A_u, 1^A_v, \ldots, 8^A_u, 8^A_v\}$$
$$BBOX^B_{2D} = \{1^B_u, 1^B_v, \ldots, 8^B_u, 8^B_v\}$$

The array forms of $BBOX^A_{2D}$ and $BBOX^B_{2D}$ differ only because certain pixel coordinates differ, such as $(1^A_u, 1^A_v) \neq (1^B_u, 1^B_v)$. For the sake of the argument to come, let's assume that the dataset consists *only* of the pair of images in Figure 3.5, i.e., the first being one where the camera looks along the Z_f-axis into the flower, and the second image's camera is the same as the first one, the difference being that it has been rotated by π about Z_f.



(A) (B)

FIGURE 3.5
Two images of the same flower model as the one in Figure 3.4. The image in Figure 3.5a is the same as the one in Figure 3.4a. In Figure 3.5b the camera has been rotated by π about the Z_f-axis. Note that the bounding box has *not* moved, only the camera.

Now, let's imagine the model during training. Its prediction for a pixel i is denoted by (i^p_u, i^p_v). The model receives as input one of the images from the described dataset, let's assume it is the image in Figure 3.5a. From this image alone, the model would likely "learn" (by performing gradient descent) that the pixels $(1^A_u, 1^A_v)$, $(2^A_u, 2^A_v)$, $(5^A_u, 5^A_v)$ and $(6^A_u, 6^A_v)$ are positioned farther down on the image plane than the pixels $(3^A_u, 3^A_v)$, $(4^A_u, 4^A_v)$, $(7^A_u, 7^A_v)$ and $(8^A_u, 8^A_v)$, and that the next time the model sees an image with similar features, it should conform its predictions to satisfy this observation. What will happen when the model sees the image in Figure 3.5b? If we are optimistic, we might expect it to identify the same features as the ones present in the first image: an orange center and white petals with purple accents. If this is the case, then under reasonable assumptions, the model will predict an ordered set of points where the pixels $(1^p_u, 1^p_v)$, $(2^p_u, 2^p_v)$, $(5^p_u, 5^p_v)$ and $(6^p_u, 6^p_v)$ are positioned quite low on the image plane and the pixels $(3^p_u, 3^p_v)$, $(4^p_u, 4^p_v)$, $(7^p_u, 7^p_v)$ and $(8^p_u, 8^p_v)$ are be positioned quite high on the image plane. This is a reasonable assumption since this is what the model learned when it saw the first image. However, if we look at where the ground truth points are located in Figure 3.5b, we can conclude that such a prediction will be heavily penalised if we used the average squared distance. For example, the pixel $(2^B_u, 2^B_v)$ is placed in the top right of the image plane, not near the bottom which is what was predicted. The loss will, in a way, indicate to the model that the predictions should have been flipped, i.e., the pixels predicted to be low on the image plane should have been higher and the pixels predicted to be high on the image plane

should have been lower. At each iteration of seeing these two images, the model will receive conflicting information and one likely possibility is that the model will simply start predicting all the pixels to be close to the image plane's center, so as to satisfy the conflicting conditions.

We could expect the model to learn to differentiate between the more fine-grained features in the images since they are not *completely* identical, but there is no guarantee that this will happen, and it is even more unlikely if we consider a pair of images of a flower that is inherently symmetrical and appears practically identical if the camera is rotated by π about Z_f, take for example the sunflower in Figure 3.1.

To counter this, we could implement the already presented solution of rotating the bounding box in 3D space and then reprojecting it. However, this solution has two main problems. The first one, which we have already seen, is that it can, and will, break the definition of what a bounding box is if the lower and upper quadrilaterals are not perfect squares, which isn't always the case. The second problem is that it is not very efficient. Two images of the same flower will be associated to the same set of rotations in 3D space of the bounding box, since the $BBOX_{3D}$ set is the same for all images of the same flower model. This means that, unless more memory is allocated during the training process to save each set of rotations once it has been computed, the same sets of rotations will end up being computed several times since they are associated to every flower model, not every image. These two problems are not absolute deal-brakers, but I will now present my solution that does not rely on rotations in 3D space and allows a model to perform better than if the average squared distance as a loss metric was used.

3.2.3 ROTATIONS IN 2D SPACE

My solution draws inspiration from how object symmetry was handled in the definition of a loss function in [22]. In their work, they attempt to perform 6D pose estimation in 3D space and define two loss function which are averaged to give the final loss function. The first one is called PoseLoss (PLOSS), and it measures, for each 3D object, the average squared distance between the ground truth points and their corresponding predictions. Note that "PLOSS" in this paragraph refers to the PLOSS from [22]. Outside of this paragraph, "PLOSS" refers to the loss function defined in Section 3.2.1. The second one is called ShapeMatch-Loss (SLOSS), and it computes, for each 3D object, the average squared distance between each predicted point and the closest ground truth point. That is, for every predicted point, whichever ground truth point produces the smallest loss when matched with it, will be the one used for computing the loss associated to that specific predicted point. Their final loss is then: $\text{LOSS} = \frac{\text{PLOSS}}{2} + \frac{\text{SLOSS}}{2}$. I should add that it is not simply the differences between 3D points that is computed in their work, but rather the difference between the same rotation matrix applied to a predicted and a ground truth quaternion, a quaternion being a number system derived from complex numbers that is often used in 3-dimensional geometry.

What we can notice is that if we applied the SLOSS to the previous example of the pair of images from Figure 3.5, the resulting loss would be quite low when the model sees the second image, assuming that it learned from the first image. If the model predicts $(2^p_u, 2^p_v)$ to be low on the image plane, that is fine despite the fact that $(2^B_u, 2^B_v)$ is actually in the top right, since the loss taken into account will be between the predicted pixel and the ground truth one that is closest to it. Again in the context of the example with the toy dataset, let's imagine two local minima of the SLOSS function, one where the predictions always perfectly match the ground truth $BBOX^A_{2D}$ array, and the other where the predictions always perfectly match the $BBOX^B_{2D}$ array. What we can notice is that these two local minima will both have a value close to 0 when passed through the SLOSS function, and that either minima will produce the desired result: a bounding box around the flower matching very closely to the ground truth. Indeed, it doesn't matter which minima is reached, both will be equally as good.

These are not the only minima though, since the model could predict *all* points to have the same coordinates as *one* of the ground truths points in either $BBOX^A_{2D}$ or $BBOX^B_{2D}$, and still have a very

small SLOSS. The problem with the SLOSS is then that a bounding box's structure could be completely lost during training. In this work, we would like to enforce the model to predict a structured bounding box while granting some freedom as to where the specific predictions land on the image plane. In Figure 3.5a for example, if $(2_u^A, 2_v^A)$, a point on the upper quadrilateral, is predicted to be $(4_u^A, 4_v^A)$, another point on the upper quadrilateral, then it shouldn't necessarily be penalised, but if $(2_u^A, 2_v^A)$ is predicted to be $(1_u^A, 1_v^A)$, a point on the lower quadrilateral, then it *should* be penalised because differentiating between the upper and lower quadrilaterals are important if the bounding box is going to be used by a robotic arm that needs to know in which direction the flower's opening is pointing, which is the direction in which $\hat{\mathbf{z}}_b$ points.

Before discussing my proposed loss function, let me first give some illustrations that will make it easier to understand. In Figure 3.6 are presented what I define as the four rotations of a quadrilateral. The shapes are in fact rectangles but I use the term quadrilateral because it is more universal. The first quadrilateral is on the left and each point is labelled with A, B, C or D. In the second quadrilateral each point has moved along an edge by one step clockwise. The points are now labelled with a subscript 2. This is repeated two more times and the consequence is that across all four rotations, each point A, B, C and D covers each vertex on the quadrilateral exactly once.

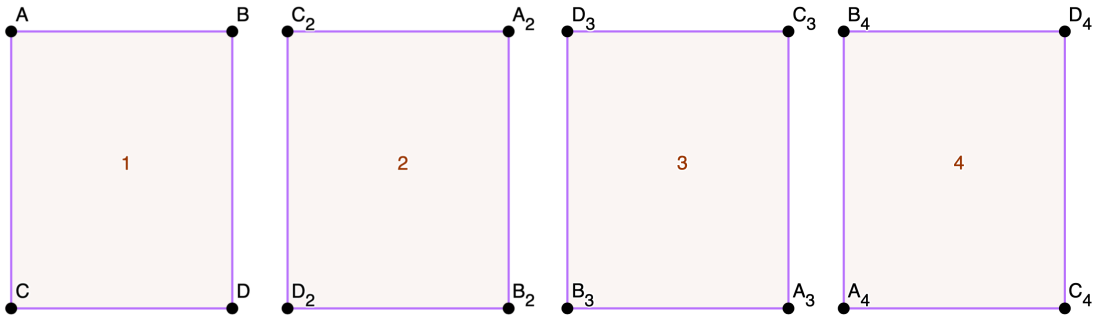


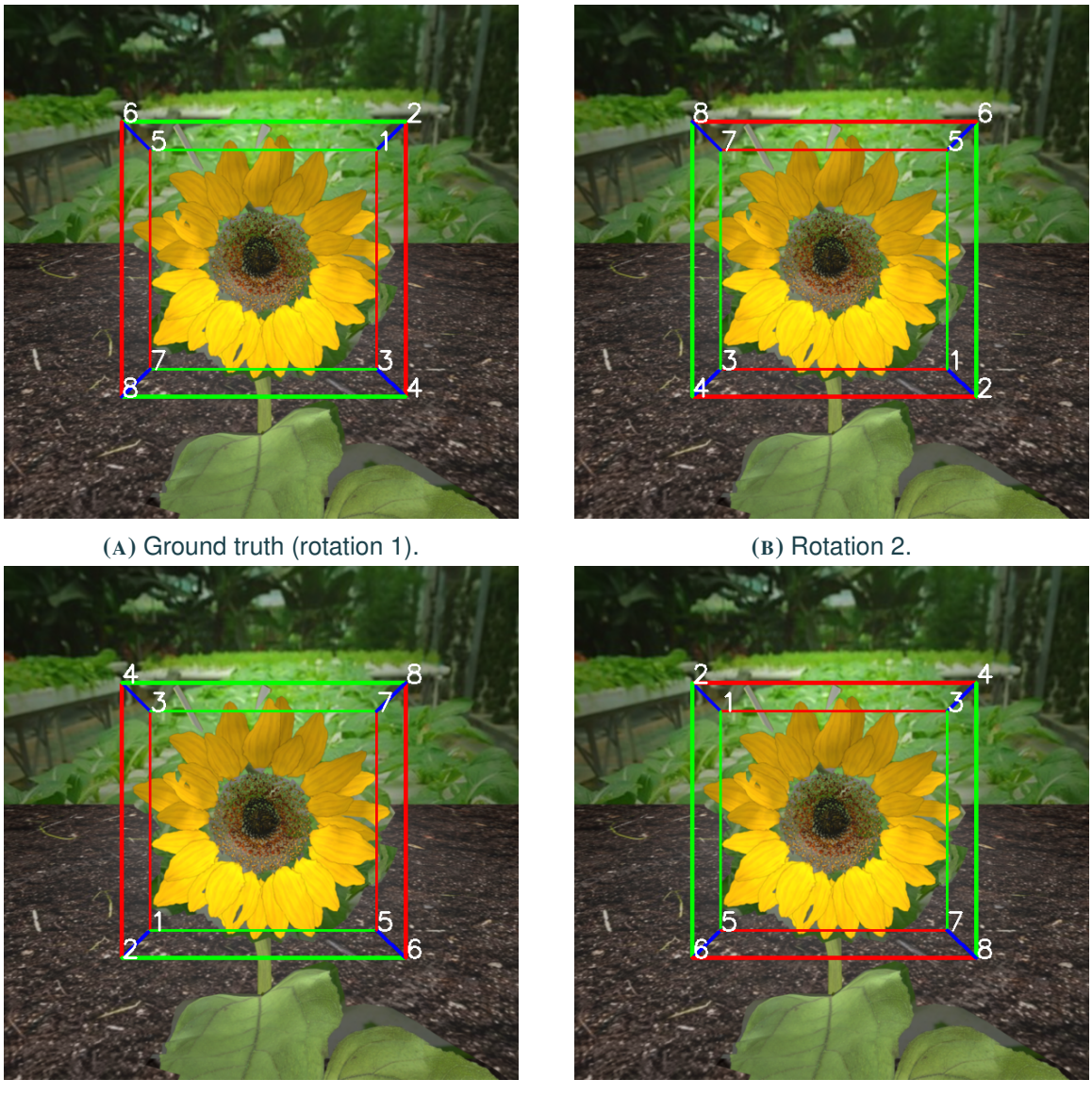
FIGURE 3.6
The 4 rotations of a quadrilateral.

However, in this work we are dealing with 3-dimensional bounding boxes, not 2-dimensional quadrilaterals. But the observation to make is that a 3-dimensional bounding box can be defined in terms of a lower quadrilateral and an upper quadrilateral, together they enclose the flower in 3-dimensional space. This is the kind of bounding box we are used to and I will also refer to it as a "quadrilateral bounding box". When these points are mapped onto the image plane, it is simple to rotate the points of both the upper and lower quadrilateral around, we simply need to cleverly swap the pixels. When the points of the *upper* quadrilateral are rotated by one step clockwise, such as when going from the first quadrilateral to the second one in Figure 3.6, the points of the *lower* quadrilateral are also rotated by one step clockwise. This ensures that a quadrilateral bounding box on the image plane has 4 distinct rotations, which can be seen in Figure 3.7, where the upper quadrilateral is labeled with even numbers and the lower quadrilateral is labeled with odd numbers. Note that the points form pairs: 1 is always next to 2, 3 is always next to 4, 5 is always next to 6 and 7 is always next to 8. This a direct consequence of performing identical rotations to the upper and lower quadrilaterals.

A rotation of the 3D bounding box then becomes a sequence of swaps applied to the contents of the array containing the ground truth bounding box. If the ground truth bounding box in Figure 3.7a is $BBOX_{2D}$ and the rotated bounding box from Figure 3.7b is $BBOX_{2D}^R$, their respective contents will be:

$$BBOX_{2D} = \{1_u, 1_v, 2_u, 2_v, 3_u, 3_v, 4_u, 4_v, 5_u, 5_v, 6_u, 6_v, 7_u, 7_v, 8_u, 8_v\}$$
$$BBOX_{2D}^R = \{5_u, 5_v, 6_u, 6_v, 1_u, 1_v, 2_u, 2_v, 7_u, 7_v, 8_u, 8_v, 3_u, 3_v, 4_u, 4_v\}$$

The contents of $BBOX_{2D}^{R}$ match what is observed in the images: pixel 5 takes the place of pixel 1, 6 takes 2's spot, 1 takes 3's spot 2 takes 4's spot, and so on. Furthermore, for any of the four rotations, if one of the four pairs of pixels $(1, 2)$, $(3, 4)$, $(5, 6)$ or $(7, 8)$ is picked at random, the pair that will come next after a clockwise rotation will be the same for all four rotations. Take for example Figure 3.7a. If we choose the pair $(3, 4)$, the pair that comes after it in a clockwise rotation is the pair $(7, 8)$, and this is the case for all four rotations. This means that the pairs have a fixed order. This, in addition to the fact that the pixels come in pairs and that the odd pixels belong to the lower quadrilateral and the even pixels belong to the upper quadrilateral, gives the the set of four rotations a hard coded structure that can hopefully be learned by an appropriate loss function.



(A) Ground truth (rotation 1).

(B) Rotation 2.

(C) Rotation 3.

(D) Rotation 4.

FIGURE 3.7
The four rotations of a flower's ground truth 3D bounding box.

Since there are four rotations for a quadrilateral bounding box, I'll denote the set containing them by $\mathcal{R}_{\mathcal{Q}} = \{BBOX_{2D}^{R_i}\}_{i=1}^{4}$, where the $_{\mathcal{Q}}$ in $\mathcal{R}_{\mathcal{Q}}$ stands for *q*uadrilateral and the ground truth $BBOX_{2D} = BBOX_{2D}^{R_1}$ is considered the first of the four possible rotations.

The loss, which I will call the RotationLoss (RLOSS), between a predicted quadrilateral bounding box $\tilde{\mathbf{b}}$ and the ground truth $\mathbf{b}$ can then be defined as:

$$\text{RLOSS}_{\mathcal{Q}}(\tilde{\mathbf{b}}, \mathbf{b}) = \min_{\mathbf{r} \in \mathcal{R}_{\mathcal{Q}}} \frac{1}{16} \|\tilde{\mathbf{b}} - \mathbf{r}\|_2^2$$

The division is by 16 because each of the eight pixels has a U and a V component. This function will compute the average squared distance between the predicted bounding box and each of the four rotations of the ground truth bounding box, returning the smallest of the four. The hope is that a model trained to minimize it can learn the hard coded structure of a quadrilateral bounding box, while still being allowed to choose freely which of the four rotations to predict.

It now seems as though the long journey taken to find a qualified and efficient criterion measuring the distance between a prediction and the ground truth has come to an end, but this is not quite the case. If we take a look at the first figure from this section, Figure 3.1, we can see that the $\text{RLOSS}_{\mathcal{Q}}$ would still heavily penalize the case where the prediction $\tilde{\mathbf{b}}$ is the rotated bounding box in Figure 3.1b and the ground truth $\mathbf{b}$ is the bounding box in Figure 3.1a. It is impossible to not heavily penalize such a rotated bounding box because the rotations used in $\text{RLOSS}_{\mathcal{Q}}$ are modified versions of the ground truth where the pixels have simply been swapped around. Indeed, if all four rotations were converted into unordered sets of pixels, they would all be equal since they all contain the exact same pixels, it just the ordering of the pixels that differ across the rotations. We could argue that, in the specific case of Figure 3.1, this is not actually a problem since the model would learn to predict a quadrilateral bounding box whose edges are parallel to the bounds of the image, and such predictions might actually be desirable. But how can we be sure that a model will learn to predict poses in such a way? Wouldn't it be better to give the models more freedom in terms of what rotations classify as valid? This could lead to better performing models since there would maybe be a greater amount of possible minimas for the cost function to reach. Is there a way of allowing for more than just the four rotations defined by $\mathcal{R}_{\mathcal{Q}}$, still keeping the desired hard coded property of the bounding box, but whitout resorting to rotations in 3D space which we have already covered? I argue that this is possible by leaving the $\text{RLOSS}_{\mathcal{Q}}$ function practically unchanged, and instead modifying our definition of a 3D bounding box, which up until now has been based around the quadrilateral shape. I will start by explaining the approach taken and then argue in favor of it.

My proposed approach is to transform the two upper and lower quadrilaterals, converting what we have referred to as the quadrilateral bounding box, into two octagons. Together, the upper and lower octagons would form an "octagon bounding box". See Figure 3.8 for a visualization and explanation of how an octagon bounding box can be generated from a quadrilateral bounding box. Notice that the octagon bounding box can be generated from the pixels contained in the $BBOX_{2D}$ set, we do not need to perform any computations in 3-dimensional space and then attempt to project the new points back to the image plane. Since all points on the image plane are defined in terms of a U and a V component, the midpoint (m_u, m_v) of any line segment joining two pixels (i_u, i_v) and (j_u, j_v) can be easily found by computing:

$$(m_u, m_v) = \Big(\frac{i_u + j_u}{2}, \frac{i_v + j_v}{2}\Big)$$

Generating the octagon variant of a quadrilateral bounding box seems straight-forward, but why is it useful to us? The whole point of using an octagon bounding box is to try and remedy the problem we discussed where the $\text{RLOSS}_{\mathcal{Q}}$ fails to handle the case in Figure 3.1 where the ground truth quadrilateral bounding box is rotated by approximately $\frac{\pi}{4}$ about Z_f.

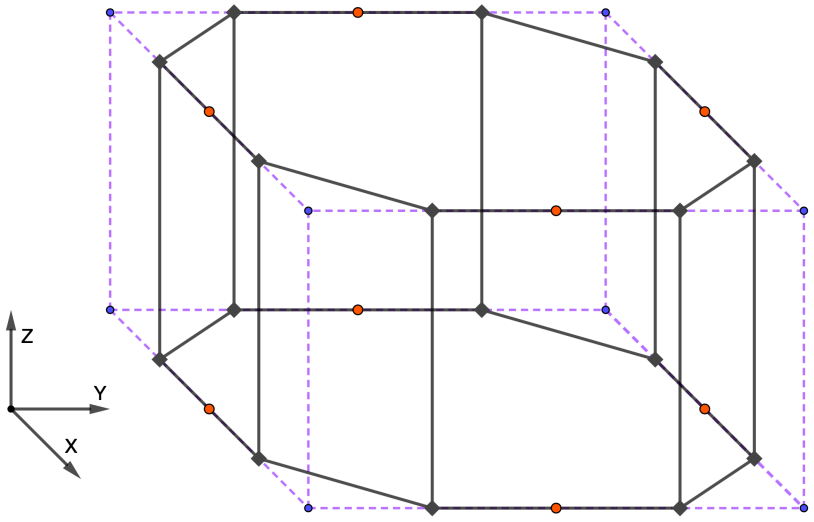
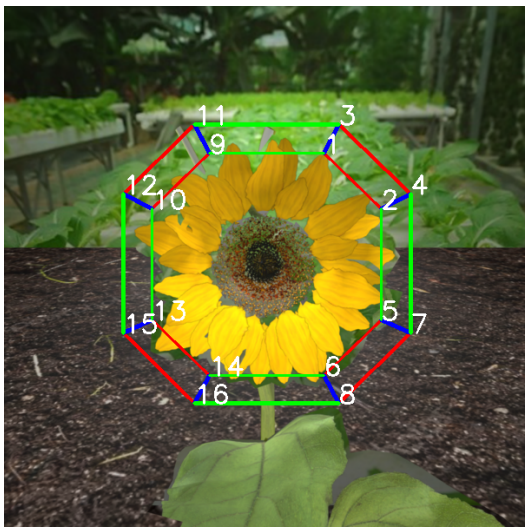
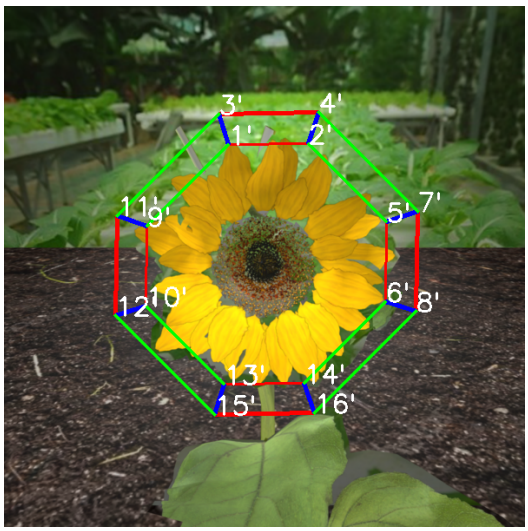


FIGURE 3.8
An octagon bounding box derived from a quadrilateral bounding box. The quadrilateral bounding box is defined by the blue circles joined by dashed purple lines. In this figure, we can imagine that the four points defining the upper quadrilateral lie on the plane $z = 3$, and the four points defining the lower quadrilateral lie on the plane $z = 0$. The orange circles are the midpoints of the eight line segments defining the upper and lower quadrilaterals. If we look at only the upper quadrilateral, then the original four edges defining it are now each split into two parts of equal length. The little black squares are the midpoints of each of these new parts. For the upper quadrilateral, since there are four edges, and each one is split in half, there will be eight midpoints, one for each of these eight "half-segments". The same is done for the lower quadrilateral and we end up with a new bounding box defined in terms of 16 points, eight for the upper octagon and eight for the lower octagon. These 16 points are joined by black lines on the diagram.



(A) The ground truth 3D bounding box in the form of an octagon bounding box.



(B) A counter-clockwise rotation of the ground truth 3D bounding box about the flower's Z_f-axis, in the form of an octagon bounding box.

FIGURE 3.9
A flower from the dataset visualized with the octagon version of its 3D bounding box. In Figure 3.9b the original quadrilateral bounding box was first rotated, just as in Figure 3.1b, then it was turned into an octagon bounding box.

Let's take a look at Figure 3.9. There, the two quadrilateral bounding boxes from Figure 3.1 have been transformed into octagon bounding boxes. Note how, despite the fact the octagon bounding box is smaller in volume than the quadrilateral bounding box, it still encloses the flower quite nicely, one might even argue that it is a better fit since flowers tend to have a circular or elliptical shape, rather than a boxy quadrilateral one. I'll call the bounding box in Figure 3.9a the "ground truth octagon bounding box", which simply refers to the octagon bounding box derived from the ground truth quadrilateral bounding box, and I'll call the bounding box in Figure 3.9b the "predicted octagon bounding box", which simply refers to the octagon bounding box derived from the predicted quadrilateral bounding box. What is interesting to note is that the ground truth and predicted octagon bounding boxes seem to overlap a lot more than the two quadrilateral bounding boxes in Figure 3.1. The argument is that the more points that are used to define the upper and lower surfaces of the bounding box, the more likely the ground truth and any of its rotations will overlap. If we imagine the upper and lower surfaces as being two circles, then any rotation of the bounding "box" will be identical to the ground truth.

Figure 3.10 displays exactly the same thing as what we saw in Figure 3.6, except now we are dealing with an octagon instead of a quadrilateral. As a result, the number of rotations gets bumped up from four to eight, and each point denoted by a letter from A to H makes its way around the octagon in a clockwise fashion and covers each of the eight vertices exactly once across all rotations. For the same reasons a quadrilateral bounding box has 4 rotations, an octagon bounding box has eight. In Figure 3.11 are presented two of the eight possible rotations of the ground truth octagon bounding box. The image in Figure 3.11a is the same as in Figure 3.9a. Note that Figure 3.11b is different from Figure 3.9b because the former is a rotation of the ground truth octagon while the latter is the octagon version of the rotated quadrilateral bounding box. Another thing to note is that all the hard-coded properties from the quadrilateral bounding box translate to the octagon version. For example, the set of pixels that corresponds to the upper quadrilateral is $\{3, 4, 7, 8, 11, 12, 15, 16\}$, and that is the case across all eight rotations. The pixels also still come in pairs such as $(1, 3)$, $(2, 4)$ and so on. The only difference in the context of octagons is that we define more points. The hope is that with more points, a point on the predicted octagon bounding box is more likely to be close to a point on the ground truth octagon (or one of its rotations). I'll demonstrate this with an example, but I will first adjust the RLOSS function to correctly deal with octagon bounding boxes.

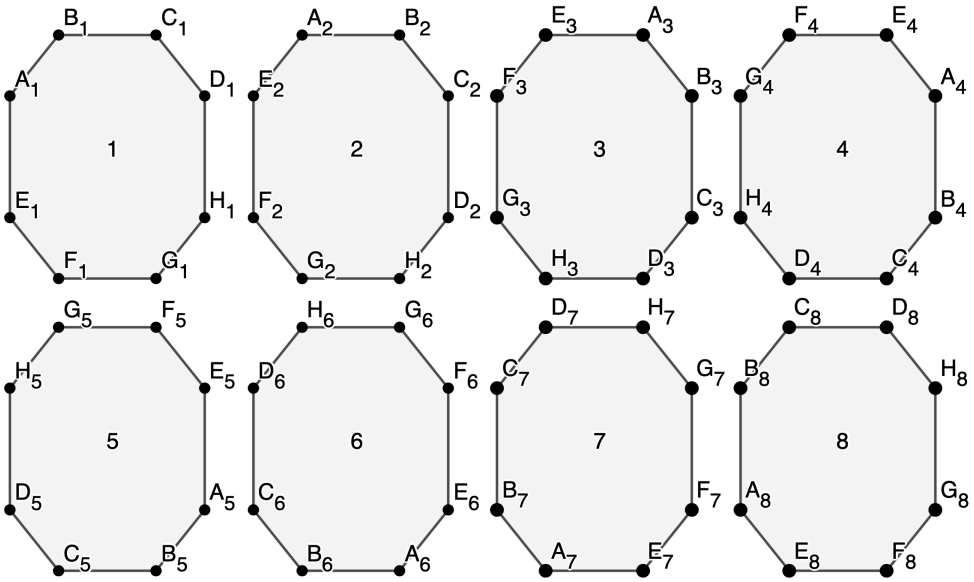
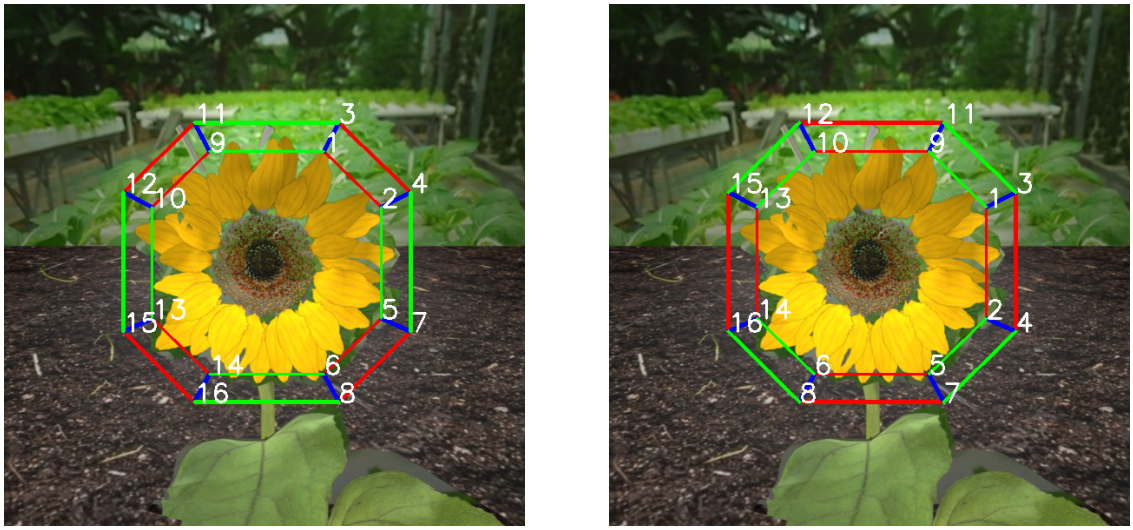


FIGURE 3.10
The eight rotations of an octagon.



(A) The ground truth octagon bounding box (rotation 1).

(B) A rotation of the ground truth octagon bounding box (rotation 2).

FIGURE 3.11
Two of the possible rotations of ground truth octagon bounding box.

Firstly, a ground truth octagon bounding box derived from the ground truth quadrilateral bounding box $BBOX_{2D}$ will be denoted by $BBO_{ct}X_{2D}$. This is isn't the prettiest notation but its use is mainly only necessary in this paragraph. In the same way that the ordered array $BBOX_{2D}^{R_2}$ is a permutation of the pixels in $BBOX_{2D}$, the ordered array $BBO_{ct}X_{2D}^{R_1}$ is a permutation of the pixels in $BBO_{ct}X_{2D}$. We'll say that the first rotation $BBO_{ct}X_{2D}^{R_1}$ is the ground truth octagon bounding box $BBO_{ct}X_{2D}$. We can define the set containing the eight rotations as: $\mathcal{R}_{\mathcal{O}} = \{BBO_{ct}X_{2D}^{R_i}\}_{i=1}^{8}$, where the $_{\mathcal{O}}$ in $\mathcal{R}_{\mathcal{O}}$ stands for *o*ctagon.

The loss, which is still called the RLOSS, between a predicted *quadrilateral* bounding box $\tilde{\mathbf{b}}$ and the ground truth *quadrilateral* bounding box $\mathbf{b}$ can then be defined as:

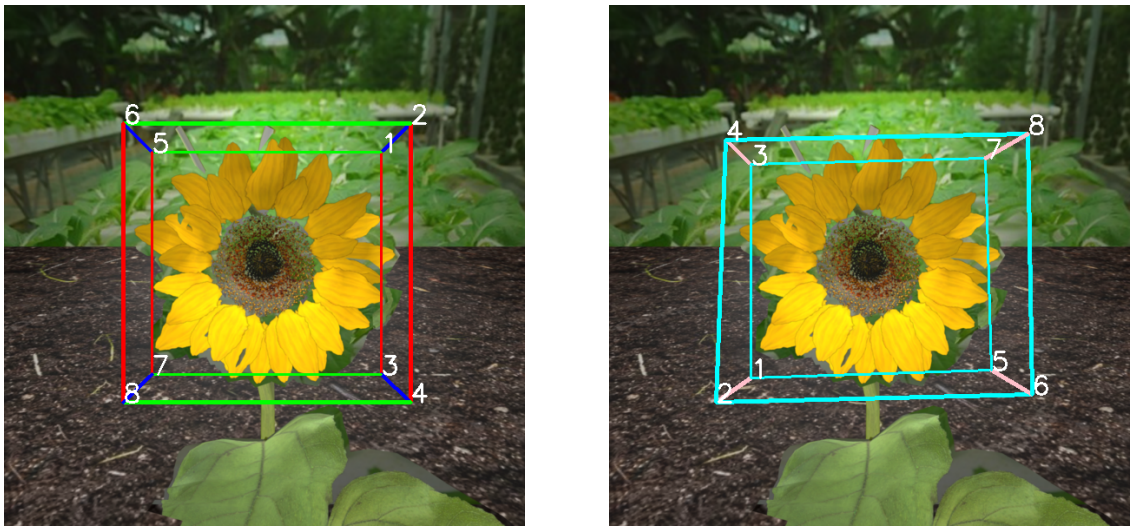
$$\text{RLOSS}_{\mathcal{O}}(\tilde{\mathbf{b}}, \mathbf{b}) = \min_{\mathbf{r}_o \in \mathcal{R}_{\mathcal{O}}} \frac{1}{32} \|\tilde{\mathbf{b}}_o - \mathbf{r}_o\|_2^2$$

The important thing to note is that $\tilde{\mathbf{b}}_o$ is the octagon version of the quadrilateral prediction $\tilde{\mathbf{b}}$. A model using $\text{RLOSS}_{\mathcal{O}}$ as a loss function is still trained to predict quadrilateral bounding boxes, but each prediction will be transformed into its octagon counterpart which will be used to compute the loss. The set $\mathcal{R}_{\mathcal{O}}$ is the set of octagon bounding box rotations derived from the ground truth quadrilateral bounding box $\mathbf{b}$. Now, both $\tilde{\mathbf{b}}_o$ and $\mathbf{r}_o$ have 32 entries since each of the sixteen pixels has a U and V component, meaning that the division is by 32 instead of 16 like it is in $\text{RLOSS}_{\mathcal{Q}}$.

For a quick comparison between $\text{RLOSS}_{\mathcal{Q}}$ and $\text{RLOSS}_{\mathcal{O}}$, let's apply it to the two bounding boxes from Figure 3.1, where $\mathbf{b}$ will be the ground truth quadrilateral bounding box from Figure 3.1a and $\tilde{\mathbf{b}}$ will be the tilted quadrilateral bounding box from Figure 3.1b. This whole discussion started when I argued that, despite $\tilde{\mathbf{b}}$ being "far" from the ground truth $\mathbf{b}$, it still qualifies as a correct bounding box. The computed losses are:

$$\text{RLOSS}_{\mathcal{Q}}(\tilde{\mathbf{b}}, \mathbf{b}) = 12{,}734.487$$
$$\text{RLOSS}_{\mathcal{O}}(\tilde{\mathbf{b}}, \mathbf{b}) = 279.600$$

To give an idea of what kind of improvement this difference represents, let's take a look at the prediction of one of the trained models in Figure 3.12. The prediction in Figure 3.12b achieves an $\text{RLOSS}_{\mathcal{Q}}$ of 114.025, which is quite low and expected since the ground truth and prediction both significantly overlap, so we consider it a valid prediction. We can leverage this computed loss by concluding that training a model using $\text{RLOSS}_{\mathcal{O}}$ enables most rotations of the ground truth $\mathbf{b}$ to be considered as valid predictions. This is because the loss (279.600) is quite close to the loss computed for a valid prediction in Figure 3.12 (114.025). And, on the contrary, the loss (279.600) is much smaller than the unnecessarily large large loss (12,734.487) computed using $\text{RLOSS}_{\mathcal{Q}}$. It is unnecessarily large because its magnitude doesn't reflect the fact that the rotated bounding box from Figure 3.1b should count as a valid prediction.



(A) The ground truth quadrilateral bounding box.

(B) The predicted quadrilateral bounding box.

FIGURE 3.12
A flower from the dataset visualized with its 3D bounding box and the prediction of one of the trained models.

This brings us to the end of this long discussion about loss functions. On a final note, regardless of which loss function was used, let's call it $\text{LOSS}(\tilde{\mathbf{b}}, \mathbf{b})$, each model was trained to minimize the following cost function:

$$\text{C}_{\text{LOSS}} = \frac{1}{n} \sum_{\mathbf{b}_i \in \mathcal{T}_s} \text{LOSS}(\tilde{\mathbf{b}}_i, \mathbf{b}_i)$$

Where $\mathcal{T}_s$ is the training set, n is its cardinality, and for an image i in the training set, $\mathbf{b}_i$ and $\tilde{\mathbf{b}}_i$ denote the ground truth and predicted quadrilateral bounding boxes respectively. In the next section I will briefly discuss some of the other design choices that can be made.

3.3 OTHER DESIGN CHOICES

3.3.1 DATA PREPROCESSING

Data preprocessing, while not exactly related to the model's design, is often used in machine learning to scale each feature dimension of the dataset to fall within a specific range, so that the features that have very large values don't unnecessarily dominate the training results purely because of their magnitude. When input features have common ranges, the gradients computed are more likely to be stable. When

dealing with images, we can think of the different features as being the three different color channels red, green and blue. In the case of RGB images though, the three color channels are already in approximately equal ranges, the minimum and maximum values possible being 0 and 255 respectively. However, scaling and normalization was tested since the pre-trained ResNet models used had been trained on normalized images from the ImageNet dataset. The per-channel mean and standard deviations of the ImageNet dataset are $\text{Mean}_{RGB} = [0.485, 0.456, 0.406]$ and $\text{STD}_{RGB} = [0.229, 0.224, 0.225]$. The normalization applied is a Z-score normalization, consisting of subtracting from every image's color channel the corresponding mean from Mean_{RGB} and then dividing by the corresponding standard deviation from STD_{RGB}. At first, I was unaware of the fact that the ResNets trained implemented normalization, so most of the experiments don't have a preprocessing step. However, ResNets use what is called batch normalization [25], which, during training, applies Z-score normalization to the activations of each layer which would, supposedly, allow each layer to learn on a more stable distribution of inputs, and thus accelerate the training of the network. For this reason, the fact that preprocessing wasn't always used can be seen as a small detail.

3.3.2 OUTPUT ACTIVATION FUNCTION

At first it was thought that the final fully connected layer of the trained models should output values between 0.0 and 1.0 by applying a sigmoid function to each neuron's output, and to then multiply the outputs by 600 to get the coordinate on the image plane. Some initial tests showed that the models trained this way always converged to a state where any output was either 0.0 or 1.0. The reason for this is not quite known, but it might have been because of the learning rate being too high, however 0.0001 isn't that big so nothing can be said for sure as it might have been due to an implementation error. Other initial tests where the model's output was not constrained to be in any given interval (i.e., using a linear activation function) showed that convergence towards appropriate pixel values in the range $[0.0, 600.0]$ happened quite quickly. For this reason, all models trained use the linear activation function for the final layer. Note that the range of pixel values in the ground truth dataset is closer to $[-30.0, 630.0]$ since some bounding boxes protrude out of the image plane's bounds.

3.3.3 NETWORK DEPTH

As was stated in Chapter 1, deeper ResNets achieve better performance on image classification tasks. For this reason, the two main ResNets tested were ResNet18 and ResNet34, where 18 and 34 indicate the number of stacked convolutional layers in each network's structure. ResNet18 is the shallowest pre-trained ResNet available for use in PyTorch and deeper networks than ResNet34 were not tested because it would simply take too long to train.

3.3.4 DATA AUGMENTATION

Data augmentation is a technique often used in machine learning to increase the number of data samples by adding slightly modified copies of these to the dataset. For example, if we are trying to solve an object classification task using labelled images, we could define a modification to be some kind of random crop of the image. The advantage is that the label assigned to this new modified version can remain unchanged. If the data sample was originally an image of a cat, then the cropped sample will still be *an image of a cat*. The problem of applying such an augmentation in this work is that cropping an image would mean that the pixels in $BBOX_{2D}$ would need to be updated according to what kind of crop it was. However, one possible augmentation could be to rotate an image by either 90, 180 or 270 degrees. Rotating the pixel coordinates contained $BBOX_{2D}$ by these three fixed angles is easy to do. A reason for possibly wanting to perform such an augmentation is that we want a trained model to be able to determine the pose of a flower, so if a flower is upside down in the input image, we'd want the model to predict that the flower is indeed upside down.

3.3.5 OPTIMIZATION

The optimizer chosen was the simple stochastic gradient descent with a learning rate of 0.0001 and a batch size of 19, therefore becoming a mini batch gradient descent. The batch size 19 was chosen because it was the largest divisor of the training set's cardinality that could be fit into the GPU's memory.

CHAPTER 4

EXPERIMENTS

In this chapter will present the different models trained and attempt to interpret the results.

4.1 EVALUATION METRICS

First, let me present the evaluation metrics used. Obviously, we could simply evaluate on the test set using the cost function C_{LOSS} defined at the end of Section 3.2 and let LOSS be one of PLOSS, $RLOSS_{\mathcal{Q}}$ or $RLOSS_{\mathcal{O}}$. However, this would simply tell us the average loss for all the samples in the test set. The next two subsections discuss two different kinds of metrics that accept a predicted bounding box as correct if the distance d between the prediction and ground truth is below a certain threshold. The distance d, as we will see, isn't necessarily limited to a 2-dimensional distance between pixels on the image plane. By using such a metric, a model's accuracy can be obtained by computing the ratio of correctly predicted bounding boxes to the total number of predictions.

4.1.1 REPROJECTION ERROR

The first option is to use something similar to the reprojection error. The reprojection error is the 2D distance $d(\mathbf{x}, \tilde{\mathbf{x}})$, where, given a camera projection matrix P [26], $\mathbf{x} = P\mathbf{X}$ is the projection of the ground truth 3D point $\mathbf{X}$ onto the image plane, and $\tilde{\mathbf{x}} = P\tilde{\mathbf{X}}$ is the projection of the predicted 3D point $\tilde{\mathbf{X}}$. Since the goal in this work is to predict 2D points on the image plane, and not 3D points, we have to modify the definition a little bit. One thing to remember is that we have access to the ground truth 3D coordinates of the points, denoted by the set $BBOX_{3D}$. We can use this set along with a predicted set $BBOX_{2D}^{P}$ to solve the PnP problem. We'll pass $(BBOX_{3D}, BBOX_{2D}^{P})$ as a set of 3D-to-2D correspondences to an OpenCV function, `solvePnP`, that solves the PnP problem. The function will return a rotation matrix and translation vector that, together, transform the coordinates of points in the "world" coordinate system to coordinates with respect to the camera's own coordinate system, i.e., it computes the optimal camera pose that would "explain" these 3D-to-2D correspondences. The pose and the camera's intrinsic properties such as its focal length are combined inside a function called `projectPoints` to create a camera projection matrix $\tilde{P}$. Using $\tilde{P}$, we can then project the ground truth 3D points to the image plane. Finally, we can compute the distance between these projected points and the ground truth 2D points.

If that was confusing, let's look at it differently. Solving the PnP problem with the ground truth correspondences between $(BBOX_{3D}, BBOX_{2D})$ would result in the "true" camera projection matrix P, since the set $BBOX_{2D}$ was originally created using $BBOX_{3D}$ and a well-defined camera during the dataset's creation. Solving the PnP problem with the correspondences between $(BBOX_{3D}, BBOX_{2D}^{P})$, note that here we are using the predicted 2D points, would indeed result in a camera projection matrix

$\tilde{P}$, but it is highly unlikely that $\tilde{P} = P$ since, for that to happen, $BBOX_{2D}^{P}$ would have to be equal to $BBOX_{2D}$, i.e., the predicted 2D points would have to be exactly identical to the ground truth 2D points. We are, in a way, transforming the set of predicted 2D points into a predicted camera projection matrix $\tilde{P}$. Let's assume we have a point $\mathbf{X}$ from $BBOX_{3D}$. To get its projection onto the image plane we apply $\tilde{\mathbf{x}} = \tilde{P}\mathbf{X}$. Remember that we also have $\mathbf{X}$'s ground truth projection onto the image plane from $BBOX_{2D}$, let's call it $\mathbf{x}$. The distance computed between these two points, $d(\mathbf{x}, \tilde{\mathbf{x}})$, where d is the L2 norm, is what we will call the reprojection error.

From here, we can define the same metric as in [27], called *2D Projection*, which accepts a predicted pose as correct if the average re-projection error, as we have just defined it, of all the points in $BBOX_{3D}$ is below 5 pixels. We'll denote the percentage of correct poses by Proj_5, i.e., it measures accuracy. We'll also evaluate at different pixel thresholds, resulting in the additions of Proj_{10} and Proj_{15}, which evaluate the *2D Projection* but with thresholds of 10 and 15 pixels respectively.

4.1.2 VERTICAL AXIS ANGLE

An additional metric was considered which assumes having access to a predicted 3D bounding box. It consists of computing the angle α between the unit vectors along the ground truth and predicted 3D bounding box's vertical axes, $\hat{\mathbf{z}}_b$ and $\tilde{\mathbf{z}}_b$ respectively. See Figure 3.2 for an illustration. Then, given a threshold t, a prediction would be considered correct if $\alpha \leq t$. If we consider the predicted 3D bounding box to be a rotation of the ground truth by any angle about $\hat{\mathbf{z}}_b$, we would have that $\alpha = 0$ implying that any rotation of the ground truth would be considered correct.

However, this metric was ultimately not used because producing a 3D bounding box from the predicted 2D points is challenging. A 3D point $\mathbf{X}$ can only be mapped to one pixel on the image plane using $\mathbf{x} = P\mathbf{X}$, where P is the camera projection matrix. On the other hand, there are infinitely many points $\mathbf{Y}$ in 3D space such that $\mathbf{y} = P\mathbf{Y}$. For this reason, it's not quite clear how transforming a predicted bounding box, i.e., a set of pixels, into its 3D counterpart could be achieved. An idea could be to attempt a least squares minimization of $\|\tilde{P}\tilde{\mathbf{X}} - \mathbf{x}\|^2$, where $\tilde{P}$, $\tilde{\mathbf{X}}$ and $\mathbf{x}$ are the predicted camera projection matrix, the 3D point to solve for, and the ground truth point respectively. However, additional constraints have to be applied since there are an infinite amount of such points $\tilde{\mathbf{X}}$. Additionally, the 3D to 2D mapping, as explained at the end of Section 2.5, consists of two steps to obtain the pixel (u,v). This would also have be taken into account.

4.2 THE MODELS

All models were tested with the same subset of the dataset. The initial first two models in the table from Figure 4.1 were trained on a smaller part of the training set, about $\frac{2}{3}$ of its final size, as those experiments were run while the dataset was still being created. For the rest of the models, the dataset $\mathcal{D}$, consisting of $65{,}574$ images, was split in the following way: Train = $0.8 \cdot \mathcal{D}$, Val = $0.1 \cdot \mathcal{D}$, Test = $0.1 \cdot \mathcal{D}$. To ensure that the three sets remained identical across all experiments, $\mathcal{D}$ was permuted using the same random seed each time. As was stated at the end of Chapter 3, different configurations of the models to train are possible. These are presented in Figure 4.1. In this chapter, models will be referenced using their ID number placed between parentheses.

Firstly, most models tested used ResNet18 because their training time for 30 epochs was around 30 hours, while for ResNet34 the training time exceeded 48 hours for 30 epochs. Next, the different loss functions discussed (PLOSS, $\text{RLOSS}_{\mathcal{Q}}$, $\text{RLOSS}_{\mathcal{O}}$) were tested, with the hypothesis being that increasing the number of points defining the bounding box, i.e., using $\text{RLOSS}_{\mathcal{O}}$ instead of $\text{RLOSS}_{\mathcal{Q}}$, would allow for more symmetries to be accepted as correct, and thus lead to better performance. Additionally, one model (7) was trained with 32 outputs, i.e., it outputs the octagon bounding box directly. This was done out of

curiosity to find out if increasing the number of outputs would impact a model's ability to learn. Since this model outputs an octagon bounding box it could not be tested on metrics that involve quadrilateral bounding boxes, e.g., the reprojection error. A few a of the models were tested with some preprocessing done to the dataset, such as (8) which performed Z-score normalization on every image. Additionally, (9-10) trained with images' value range scaled from $[0, 255]$ to $[0.0, 1.0]$.

Model List					
ID	Depth	Loss	# Outputs	# Epochs	Preprocessing
1	18	PLOSS	16	17	None
2	18	PLOSS	16	30	None
3	18	$\text{RLOSS}_{\mathcal{O}}$	16	10	None
4	18	$\text{RLOSS}_{\mathcal{O}}$	16	30	None
5	34	$\text{RLOSS}_{\mathcal{O}}$	16	30	None
6	34	$\text{RLOSS}_{\mathcal{O}}$	16	40	None
7	18	$\text{RLOSS}_{\mathcal{O}}$	32	30	None
8	18	$\text{RLOSS}_{\mathcal{O}}$	16	30	ImageNet Z-score
9	18	$\text{RLOSS}_{\mathcal{Q}}$	16	15	Scale to $[0, 1]$
10	18	$\text{RLOSS}_{\mathcal{Q}}$	16	30	Scale to $[0, 1]$
11	18	$\text{RLOSS}_{\mathcal{Q}}$	16	10	None
12	18	$\text{RLOSS}_{\mathcal{Q}}$	16	17	None
13	18	$\text{RLOSS}_{\mathcal{Q}}$	16	30	None

FIGURE 4.1
Table description of the different models. The dashed lines group together identical model configurations where only the number of epochs varies.

4.3 RESULTS

The tables in Figure 4.2 and Figure 4.3 summarize the test results. The two first columns provide the ID from Figure 4.1 and a description in the form $\{\text{Preprocessing}\}\{\text{LOSS}\}\{\text{Depth}_{\text{epochs}}\}$. Again, the rows grouped together inside the dashed lines represent identical model configurations, where only the epoch number differs and it increases the farther down the row is inside the group. Note that the columns of the table in Figure 4.2 contain the value of the cost function C_{LOSS}, but where LOSS is replaced by the loss function specified by each column header. In this chapter an "$\text{RLOSS}_{\mathcal{Q}}$ model" will be a model trained to minimize $\text{C}_{\text{RLOSS}_{\mathcal{Q}}}$. If there is no subscript $_{\mathcal{Q}}$ or $_{\mathcal{O}}$, the reader can safely assume that I am grouping together the models, i.e., a reference to "RLOSS models" would be all models trained using $\text{RLOSS}_{\mathcal{Q}}$ or $\text{RLOSS}_{\mathcal{O}}$ as loss function.

The first ting to notice in Figure 4.2 is that RLOSS models achieve a PLOSS value that is more than twice that of PLOSS models. This could be seen as normal since a model is expected to perform well on the cost function it was trained to minimize. We can also interpret this to mean that the RLOSS models consistently predict bounding boxes that are one of the four (or eight) possible rotations, i.e., by not training with PLOSS and by not being constrained to predict *one* ground truth rotation, the RLOSS models make predictions that are the rotated versions of the ground truth. Because of this, the PLOSS models achieve smaller PLOSS values than RLOSS models, taking the example of (1) achieving a PLOSS of 8004 compared to (12) achieving a PLOSS of $20{,}471$ in the table from Figure 4.2. However, smaller PLOSS values do not translate to better accuracy as (12) achieves better accuracies than (1) for every single Proj metric in the table from Figure 4.3. It can therefore be concluded that the minimas reached by RLOSS models lead to better performance than the minimas reached by PLOSS models.

Model Performances Using C_{LOSS}				
ID	Description	PLOSS	$RLOSS_{\mathcal{Q}}$	$RLOSS_{\mathcal{O}}$
1	$PLOSS_{\mathcal{Q}}18_{17}$	8004	4155	2719
2	$PLOSS_{\mathcal{Q}}18_{30}$	8694	4382	2840
3	$RLOSS_{\mathcal{O}}18_{10}$	20,469	4096	2147
4	$RLOSS_{\mathcal{O}}18_{30}$	21,216	2135	164
5	$RLOSS_{\mathcal{O}}34_{30}$	21,395	2153	136
6	$RLOSS_{\mathcal{O}}34_{40}$	21,266	2148	122
7	$RLOSS_{\mathcal{O}}18_{30}$	-	-	106
8	Z-score $RLOSS_{\mathcal{O}}18_{30}$	21,353	1638	158
9	Scale $RLOSS_{\mathcal{Q}}18_{15}$	20,705	390	227
10	Scale $RLOSS_{\mathcal{Q}}18_{30}$	21,544	150	83
11	$RLOSS_{\mathcal{Q}}18_{10}$	20,357	675	392
12	$RLOSS_{\mathcal{Q}}18_{17}$	20,471	320	191
13	$RLOSS_{\mathcal{Q}}18_{30}$	20,626	175	96

FIGURE 4.2
Table description of the cost function C_{LOSS} evaluated for the different loss functions (PLOSS, $RLOSS_{\mathcal{Q}}$, $RLOSS_{\mathcal{O}}$).

The next thing to notice, which surprised me, is that a model like (13), which is an $RLOSS_{\mathcal{Q}}$ model, performed better on the $RLOSS_{\mathcal{O}}$ metric than (4), an $RLOSS_{\mathcal{O}}$ model, the former achieving a loss of 96, while the latter's loss is 164. Additionally, (13) achieved a result of 175 on $RLOSS_{\mathcal{Q}}$, which is much better than the 2135 achieved by (4). Other than the loss function, the two models' configurations are identical. A likely reason for $RLOSS_{\mathcal{O}}$ models performing poorly on the $RLOSS_{\mathcal{Q}}$ metric is that $RLOSS_{\mathcal{O}}$ models can make predictions such as the slightly rotated one in Figure 3.1b. As we already explained at the end of Section 3.2.3, the $RLOSS_{\mathcal{Q}}$ value would be very large (12,734) for such a prediction, while the $RLOSS_{\mathcal{O}}$ value would be relatively small (279). However, the reason as to why $RLOSS_{\mathcal{Q}}$ models have the advantage for the $RLOSS_{\mathcal{O}}$ metric is not quite clear. Regardless, my assumption that using octagon bounding boxes in the loss function would lead to better performance has now been proven to not be entirely true. Looking at the table in Figure 4.3, we can see that in terms of the $Proj_5$ metric, (13) achieves an accuracy of 9.51% and (4) achieves an accuracy of 5.32%. One could argue that both values are so small that the difference is insignificant. However, the images used are 600×600, which is quite a large dimension, for that reason the $Proj_{10}$ and $Proj_{15}$ metrics were included. In terms of the $Proj_{15}$ metric, (13) achieves an accuracy of 80.71% and (4) achieves an accuracy of 47.08%. The ratio of these values is still approximately the same as in the case of $Proj_5$, but we can now admit that 80.71% is far better than 47.08%. An explanation might be that transforming quadrilateral bounding boxes into their octagon counter parts is not that effective and that allowing for more rotational symmetries would give too much freedom to the models in terms of how many rotations they can predict. We could conclude that limiting a loss function to four possible valid rotations instead of eight is better because such a constraint helps a model decide which rotation to predict, instead the model getting "confused" because it has a larger pool of possible rotations to choose from. We could also argue that four is somewhere very close to the optimal number of valid rotations since limiting this number to one, as is the case with PLOSS models, results in poor reprojection performance, as shown in Figure 4.3. Indeed RLOSS models perform better than PLOSS models across all reprojection metrics, taking the example of the $Proj_5$ metric, where (13) achieves an accuracy of 9.51 and (2) achieves an accuracy of 0.15.

Next, there is a clear improvement when using deeper networks. When comparing (4) and (5) in Figure 4.2, which only differ in terms of the depth of the network, the latter achieves a better $RLOSS_{\mathcal{O}}$ value (136) than the former (164). In Figure 4.3, (5) achieves a better $Proj_5$ accuracy of 10.15% than (4),

which is at 5.32%. (4) catches up at Proj_{15} with 47.08% compared to (5)'s 53.01%. However, achieving a high accuracy at Proj_5 is arguably more difficult than at Proj_{15} because of the smaller pixel constraint. Additionally, when (5) goes to (6) by adding 10 training epochs for a total of 40, the three accuracies go up, noting that the Proj_5 accuracy goes up to 15.90%. This leads to the conclusion that training deep networks has clear advantages, one of the main reasons being that they can learn more complex and fine-grained image features.

Model Performances Using Reprojection					
ID	Description	Proj_5	Proj_{10}	Proj_{15}	AvgReErr
1	$\text{PLOSS}_{\mathcal{Q}}18_{17}$	0.44	8.15	19.33	74.28
2	$\text{PLOSS}_{\mathcal{Q}}18_{30}$	0.15	4.49	13.54	78.17
3	$\text{RLOSS}_{\mathcal{O}}18_{10}$	0.06	1.18	4.33	77.18
4	$\text{RLOSS}_{\mathcal{O}}18_{30}$	5.32	32.06	47.08	45.47
5	$\text{RLOSS}_{\mathcal{O}}34_{30}$	10.15	41.47	53.01	43.27
6	$\text{RLOSS}_{\mathcal{O}}34_{40}$	15.90	47.37	55.24	41.99
7	$\text{RLOSS}_{\mathcal{O}}18_{30}$	-	-	-	-
8	Z-score $\text{RLOSS}_{\mathcal{O}}18_{30}$	5.76	35.08	52.43	38.22
9	Scale $\text{RLOSS}_{\mathcal{Q}}18_{15}$	0.77	16.55	45.57	23.55
10	Scale $\text{RLOSS}_{\mathcal{Q}}18_{30}$	12.10	61.08	83.71	11.84
11	$\text{RLOSS}_{\mathcal{Q}}18_{10}$	0.39	6.95	23.81	33.45
12	$\text{RLOSS}_{\mathcal{Q}}18_{17}$	1.03	21.92	52.72	20.88
13	$\text{RLOSS}_{\mathcal{Q}}18_{30}$	9.51	54.22	80.71	12.57

FIGURE 4.3

Table description of the models' performances using the reprojection accuracy metrics from Section 4.1.1. The final column, AvgReErr, contains the average reprojection error of all images in the test set.

The previous comparison was done between $\text{RLOSS}_{\mathcal{O}}$ models. However, as we found out, $\text{RLOSS}_{\mathcal{Q}}$ models perform substantially better. Model (13), an 18-depth $\text{RLOSS}_{\mathcal{Q}}$ model trained for 30 epochs, performs better than model (6), a 34-depth $\text{RLOSS}_{\mathcal{O}}$ model trained for 40 epochs. In Figure 4.2, (13) outperforms (6) in every single metric, and in Figure 4.3, (6) only has the advantage in Proj_5, with 15.90% compared to 9.51%, but (13) takes the lead in Proj_{10} and Proj_{15}, with 54.22% compared to 47.37% for Proj_{10} and 80.71% compared to 55.24% for Proj_{15}. This enforces the claim that $\text{RLOSS}_{\mathcal{Q}}$ outperform $\text{RLOSS}_{\mathcal{O}}$ models, however, we should keep in mind that the high accuracy achieved by (6) in Proj_5 holds some weight since it is difficult to achieve a good accuracy for such a small pixel threshold. The reason for the deeper network performing better on the Proj_5 metric, then getting overtaken on the Proj_{10} and Proj_{15} metrics is not quite clear to me.

As was said in Section 3.3.1, data preprocessing can speed up the training time. In Figure 4.2, if we look at model (10), where the dataset images were scaled to $[0.0, 1.0]$, it achieves smaller $\text{RLOSS}_{\mathcal{Q}}$ and $\text{RLOSS}_{\mathcal{O}}$ values than its non-scaled counterpart (13). In Figure 4.3 (10) also does very well, achieving accuracies of 12.10%, 61.08% and 83.71% for Proj_5, Proj_{10} and Proj_{15} respectively, while (13) achieves the values of 9.51%, 54.22% and 80.71% for the same metrics. This difference might be solely due to speedup in training time, i.e., with 30 epochs, (10) outperforms (13) because the former doesn't need to see the data as many times as the latter to reach the same performance. Another possibility might result from the fact that the pre-trained ResNets were trained on normalized data, meaning that when (13) sees images with values in the range $[0, 255]$ at the beginning of its training, it takes a few iterations before the weights can adapt to this difference. On the other hand, when (10) sees images with values in the range $[0.0, 1.0]$ at the beginning of its training, any adjustment necessary is minimal as the range $[0.0, 1.0]$ is quite close to what the pre-trained ResNets are used to ($[-1.0, 1.0]$). Furthermore, (8) has a

slight advantage over (4), the two only differing by the fact that the data seen by (8) was normalized using the Z-score normalization using ImageNet's mean and standard deviation. In Figure 4.3, the accuracies achieved by (8) are slightly above those achieved by (4). For these reasons, scaling and normalizing data has advantages, which is likely due to training speedup.

Another interesting result is that the PLOSS models trained had their performance decrease when going from 17 to 30 epochs. The metrics in the table Figure 4.2 and Figure 4.3 are consistently better when the number of epochs trained is 17. The reason for this might be that the models using PLOSS as loss function start to overfit to the training set, seeing as though they were trained on a smaller part of the dataset. However, nothing can be said for sure. If we look at the RLOSS models, on the other hand, whenever the number of epochs trained goes up, so does the accuracy. These RLOSS models were trained on the whole 80% of the final dataset, so it might be the case that if they were trained on an even smaller part the dataset, their performance would also decrease when the epoch number goes from 17 to 30. This could have been tested to verify the validity of the claim, however, time constraints made it difficult to perform all the experiments. It would have been interesting, for example, to see what kind of performance a 34-depth ResNet trained with $\text{RLOSS}_{\mathcal{Q}}$ could achieve, as we have seen that both deep networks and $\text{RLOSS}_{\mathcal{Q}}$ models both perform very well. We've also seen that preprocessing data such as scaling and normalizing can speed up training, but it would have been interesting to determine experimentally if either of these two methods has a clear advantage on the other.

Finally, model (7) is an $\text{RLOSS}_{\mathcal{O}}$ model and has 32 outputs. This experiment was done to find out if 32 outputs was too much to handle and would result in meaningless outputs. It could not be tested on quadrilateral or reprojection metrics as it outputs an octagon bounding box and we do not have the ground truth 3D octagon bounding box points, but, as can be seen in the table from Figure 4.2, it does perform quite well on the $\text{RLOSS}_{\mathcal{O}}$ metric, meaning that 32 outputs was not an issue for the model during training.

4.4 SAMPLE PREDICTIONS

I will end this chapter by showing some samples from the models' predictions.

(A) (B) (C)

FIGURE 4.4
Predictions from the models (4), (6) and (8) respectively.

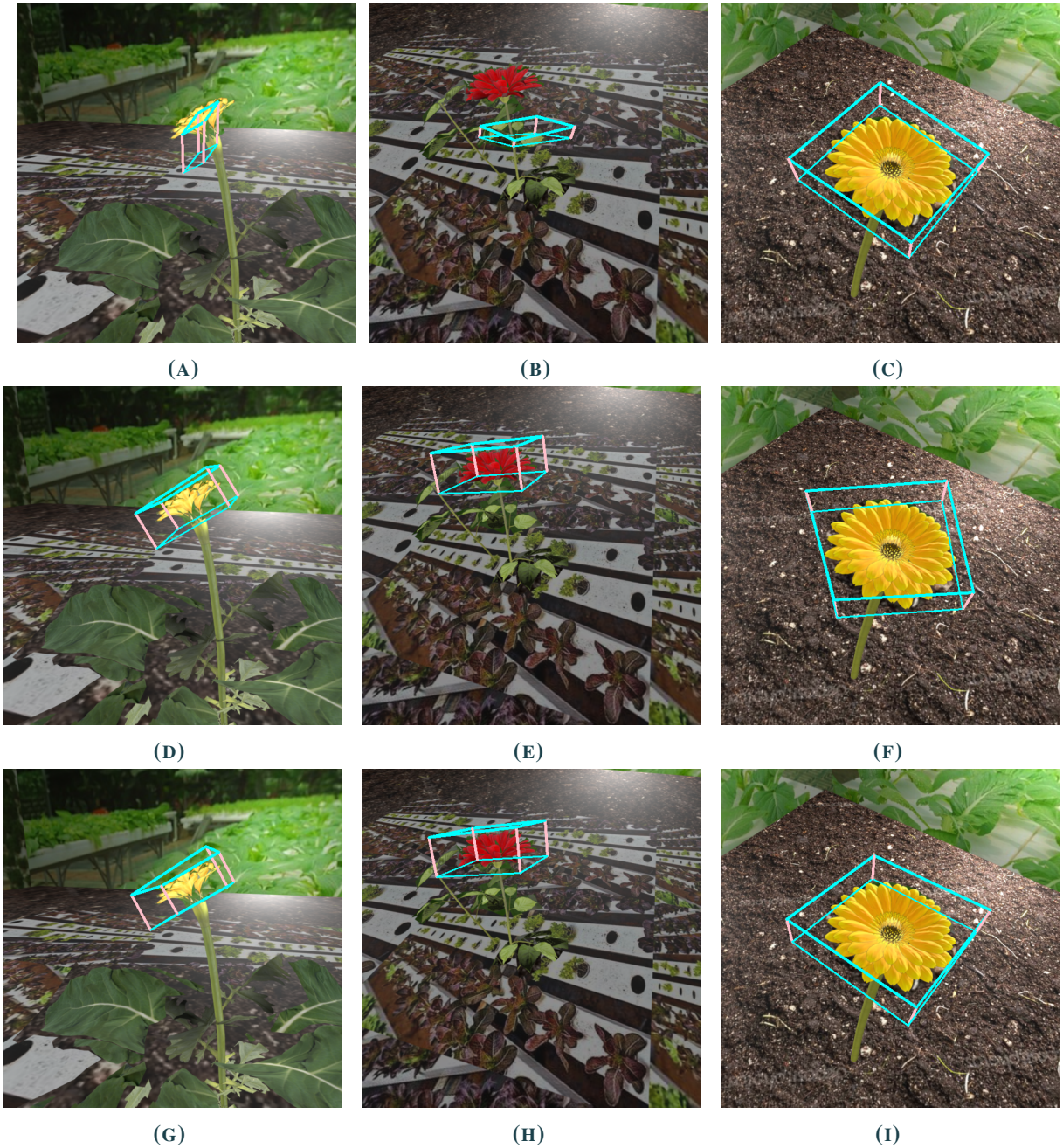
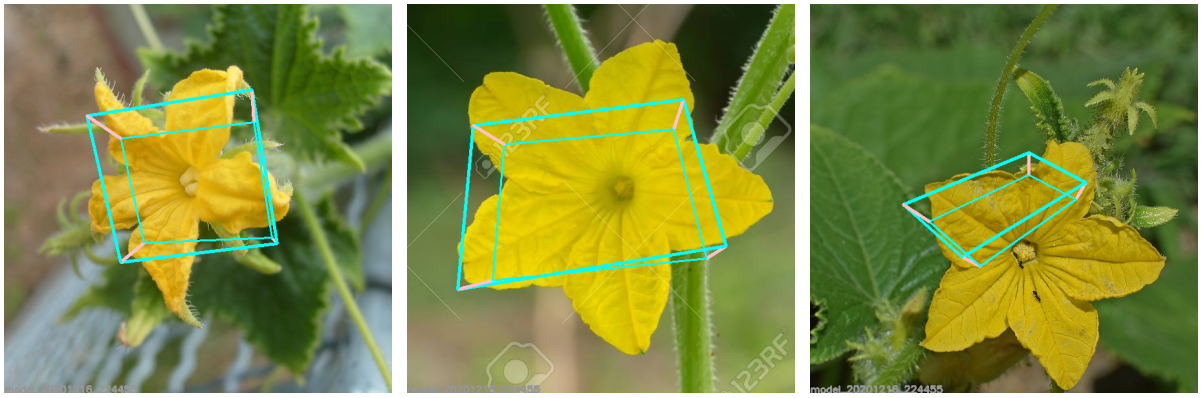
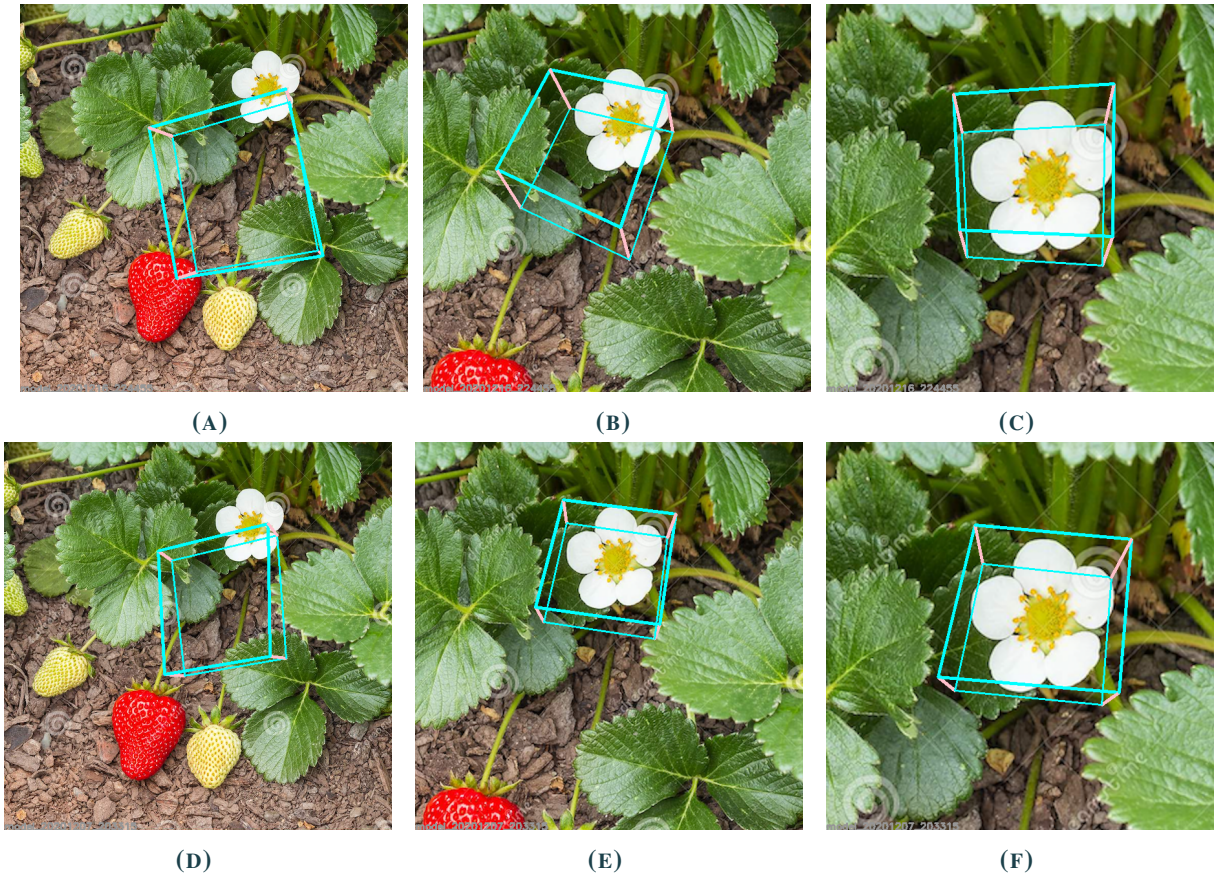


FIGURE 4.5
Predictions on the synthetic test set with three different models. The top three are made with the 18-layer PLOSS model (1). The middle three are made with the 34-layer RLOSS$_{\mathcal{O}}$ model (6). The bottom three are made with the 18-layer RLOSS$_{\mathcal{Q}}$ model (10).



(A) (B) (C)

FIGURE 4.6
Some predictions on real images of a cucumber flower, whose synthetic model is present in the dataset. The predictions are made using model (10) from Figure 4.1.



(A) (B) (C)

(D) (E) (F)

FIGURE 4.7
Some predictions on real images of a strawberry flower, whose synthetic model is present in the dataset. The top three predictions are made using model (10) from Figure 4.1, the bottom three are made using model (6). The images are progressively cropped to make the flower appear bigger. Both models perform better on images where the flower is bigger relative to the image's size, however, the 34-layer model (6) performs better than the 18-layer model (10).

CHAPTER 5

CONCLUSION

This work presented an attempt at estimating the pose of a flower. This was done using machine learning models trained on a synthetic dataset and whose loss functions centered around the challenge of accounting for a flower's rotational symmetries. The dataset creation deals heavily with synthetic 3D models and 3-dimensional geometry. The problem of accounting for a flower's rotational symmetries was closely looked at. The goal when designing an appropriate loss function to be used by the machine learning models was to consider rotations of the bounding box as valid predictions. The main idea was that a model should not be unnecessarily penalised when predicting a rotated version of the ground truth bounding box since it can still be considered correct.

The first proposed loss function was a simple average squared distance metric to serve as a baseline. Then, rotations in 3D space were studied but these had limitations such as breaking a bounding box's definition and being inefficient. Finally, rotations in 2D space were examined and two possible solutions resulted from it. The first considers a bounding box as a pair of quadrilateral surfaces enclosing a flower. With such a definition, a bounding box can be thought of as having 4 rotations and the proposed loss function would consider any of these as valid. This was thought to be too limiting, so a variation allowing for eight rotations was proposed. In this new setting, a bounding box would be seen as a pair of octagon surfaces enclosing a flower. The hypothesis was that allowing for more rotations would lead to better performing models.

As was seen when interpreting the results of the different experiments, the hypothesis of more rotations leading to better performance was concluded to be largely false, as limiting the number of rotations to four likely constrained the model in a positive way, rather than leaving it confused with too many possible rotations to choose from. On the other hand, using only one rotation was likely too limiting because of the symmetrical property of the flowers in the dataset. Therefore, using four rotations in the loss function was concluded to be the optimal trade-off between too many and not enough rotations.

Additionally, improvements could be made to a model's performance by using deeper networks. This is no surprise as deeper networks can learn more fine-grained image features. Improvements could also be made by preprocessing data, either by scaling the image values to $[0.0, 1.0]$ or by applying Z-score normalization. Both of these methods bring the input images closer to what pre-trained ResNets are used to seeing, since they have been trained on normalized images. This lead us to conclude that the performances enabled by preprocessing were likely due to a ResNet not having to adjust to a larger $[0, 255]$ interval at the beginning of its training.

A lot of experiments were carried out, however many had to be left out because of time constraints. As was said previously, training a model on rotated images by performing data augmentation would have been interesting to try. Furthermore, entirely different approaches to the rotational symmetry problem

might exist which could have led to different loss functions. It would have also been possible to test the loss functions designed, but adjusting them to use an average absolute distance instead of a squared one, to see if such a modification would have led to any changes in performance. A model capable of predicting the poses of multiple flowers in the same image would also be interesting to try. Finally, the dataset used to train the models is synthetic meaning that the performance on real images can be subpar. Possible solutions to this might be to balance the dataset with manually labelled real images or to leave them unlabelled and attempt to use semi-supervised learning, but this approach is mainly used for classification tasks. We also saw that models tend to perform better on real images when the flower takes up more space on the image plane. A solution to accurately predicting the pose of smaller flowers might be to crop a window that is centered on the flower's center and pass this window onto the model for prediction.

BIBLIOGRAPHY

[1] Konstantinos G Liakos et al. 'Machine Learning in Agriculture: A Review'. In: *Sensors* 18.8 (2018), p. 2674.

[2] Efthimia Mavridou et al. 'Machine Vision Systems in Precision Agriculture for Crop Farming'. In: *Journal of Imaging* 5.12 (2019), p. 89.

[3] Zhong-Qiu Zhao et al. 'Object Detection With Deep Learning: A Review'. In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232.

[4] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[5] J. Deng et al. 'ImageNet: A Large-Scale Hierarchical Image Database'. In: *CVPR09*. 2009.

[6] Sinno Jialin Pan and Qiang Yang. 'A Survey on Transfer Learning'. In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[7] Philipe A Dias, Amy Tabb and Henry Medeiros. 'Apple Flower Detection using Deep Convolutional Networks'. In: *Computers in Industry* 99 (2018), pp. 17–28.

[8] Dor Oppenheim, Yael Edan and Guy Shani. 'Detecting Tomato Flowers in Greenhouses Using Computer Vision'. In: *International Journal of Computer and Information Engineering* 11.1 (2017), pp. 104–109.

[9] Umme Fawzia Rahim and Hiroshi Mineno. 'Tomato Flower Detection and Counting in Greenhouses Using Faster Region-Based Convolutional Neural Network'. In: *Journal of Image and Graphics* 8.4 (2020).

[10] Zhenglin Wang, James Underwood and Kerry B Walsh. 'Machine vision assessment of mango orchard flowering'. In: *Computers and Electronics in Agriculture* 151 (2018), pp. 501–511.

[11] Manya Afonso et al. 'Detection of Tomato Flowers from Greenhouse Images Using Colorspace Transformations'. In: *EPIA Conference on Artificial Intelligence*. Springer. 2019, pp. 146–155.

[12] JongYoon Lim et al. 'Deep Neural Network Based Real-time Kiwi Fruit Flower Detection in an Orchard Environment'. In: *arXiv preprint arXiv:2006.04343* (2020).

[13] Jared Strader et al. 'Flower Interaction Subsystem for a Precision Pollination Robot'. In: *arXiv preprint arXiv:1906.09294* (2019).

[14] Dawson-Haggerty et al. *Trimesh*. Version 3.8.11. 28th Sept. 2020. URL: https://trimsh.org/.

[15] Matl et al. *Pyrender*. Version 0.1.43. 30th May 2020. URL: https://pyrender.readthedocs.io/en/latest/.

[16] Paolo Cignoni et al. 'MeshLab: an Open-Source Mesh Processing Tool'. In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara and Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[17] Blender Online Community. *Blender - a 3D modelling and rendering package*. Version 2.90.1. Blender Institute, Amsterdam: Blender Foundation, 23rd Sept. 2020. URL: http://www.blender.org.

[18] G. Bradski. 'The OpenCV Library'. In: *Dr. Dobb's Journal of Software Tools* (2000).
[19] Caner Sahin et al. 'A Review on Object Pose Recovery: from 3D Bounding Box Detectors to Full 6D Pose Estimators'. In: *Image and Vision Computing* (2020), p. 103898.
[20] Alberto Crivellaro et al. 'A Novel Representation of Parts for Accurate 3D Object Detection and Tracking in Monocular Images'. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 4391–4399.
[21] Mahdi Rad and Vincent Lepetit. 'BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth'. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 3828–3836.
[22] Yu Xiang et al. 'PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes'. In: *arXiv preprint arXiv:1711.00199* (2017).
[23] Georgios Georgakis et al. 'Matching RGB Images to CAD Models for Object Pose Estimation'. In: *arXiv preprint arXiv:1811.07249* 2 (2018).
[24] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.
[25] Sergey Ioffe and Christian Szegedy. 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. In: *arXiv preprint arXiv:1502.03167* (2015).
[26] Wikipedia. *Camera matrix*. Wikimedia Foundation. 2020. URL: https://en.wikipedia.org/wiki/Camera_matrix (visited on 21st Dec. 2020).
[27] Eric Brachmann et al. 'Uncertainty-Driven 6D Pose Estimation of Objects and Scenes from a Single RGB Image'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3364–3372.