An introduction to SurfEmb and the theory behind, and an exploration of possible improvements

Adrian Gjertsen

Coordinator
Olav Egeland

Project Thesis

Department of Mechanical Engineering
NTNU
Norway
20.12.2022

Abstract

This thesis goes over the basic concepts that are required to understand how SurfEmb and many other methods within the 6DOF pose estimation field function on a basic level. Initially, the basics of image formation is covered. The current state of the pose estimation field is then covered and the main challenges are presented. A basic introduction is given of Deep Learning and Convolutional Neural Networks to see how it functions and how the paradigm shift has changed the computer vision field. The SurfEmb method itself is then explained using the previously covered theory. Finally some alterations of SurfEmb are experimented with and results presented. Alterations are in the form of adjustments to hyperparameters, and the introduction of an early termination of their RANSAC scheme. Combining the altered parameters and the early termination gives up to 68% decrease in run-time, with little to no loss in accuracy on a limited dataset. The early termination with unchanged parameters is a free run-time improvement, shown to give up to 39% reduction in run-time.

Contents

Chapter 1

Introduction

The 6DOF pose estimation problem is the problem of finding the rotation and translation of a target object from one or multiple images. It is assumed that the target object is saved as a 3D model to the computer with a local coordinate system. The task is to find the transformation relating the object coordinate system as configured in the scene to the cameras coordinate system.

The problem has subclasses with differing data availability. Often they are distinguished by the availability of depth information and multiple views. Usually either an RGB or RGBD camera is used, where the latter uses a separate depth sensor. In some cases there are multiple cameras taking images of the same scene, giving more information about the relative locations of objects. Different methods focus on different versions of the problem. The hardest problem is the single-view RGB problem as it contains the least information about the scene.

SurfEmb[9] is a method for solving the pose estimation problem using only single-view, so this is the problem variation that will be focused on in this project. SurfEmb was presented in a very recent paper, published in November 2021. It introduces a novel representation of correspondences and with it became a state of the art pose estimation method. SurfEmb was tested and scored on the BOP datasets(https://bop.felk.cvut.cz/), becoming the best overall scoring method at the time. The BOP website is a hub meant to facilitate competition between pose estimation methods by providing training sets and scoring all methods on the same test set.

SurfEmb has since been overtaken by some other methods. Currently there are 5 other methods that rank above it in overall score on the BOP website. This is because of the fields rapid evolution. Especially in the last decade, the field has rapidly evolved because of the Deep Learning paradigm. The introduction of Convolutional Neural Networks(CNN) has opened up the possibilities for many new and unique approaches. As the deep learning field itself progresses and computational power increases the pose estimation field will benefit. While some methods use different approaches, they still have much in common and must each overcome the same central problems such as visual ambiguities.

This project will take a look at SurfEmb and its solution to the 6DOF pose estimation problem using only a single-view RGB image. In order to understand how SurfEmb functions, this thesis will go through the prerequisite theory required. First it will present the basics of image formations, followed by an explanation of what a pose estimation problem is, the challenges that arise, and how it is solved with or without deep learning. It will then give an introduction to deep learning and CNNs, and see why they are so well suited for the pose estimation task. The method and novelty of SurfEmb will then be explained. Finally some possible improvements will be discussed and experimented with.

Chapter 2

Camera Theory

In order to understand the challenges of any computer vision problem, one must know the basics of image formation. This section will briefly cover the essentials of camera formation.

Cameras, and their projective transformations, can be explained with different models. The simplest of which is the pinhole model. While it is simple, it is the most essential and can model the projections of traditional cameras. For more complicated cameras, like those using a fish-eye lens, other models are required since the lens has non-linear distortions that a traditional camera does not.

2.1 The pinhole model

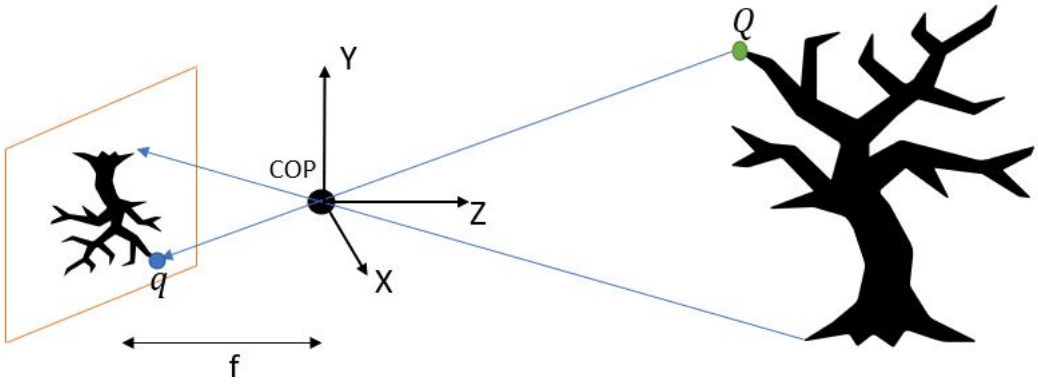


Figure 2.1: The pinhole model. The camera coordinate system is drawn. A tree projected onto an image plane. The image plane is in reality behind the COP and the projectede shape is upside down as shown. It is often flipped and drawn in front to make the math more elegant. The focal length is marked f.

The pinhole model is so named because of the assumption that every light ray from the scene goes through an infinitely small hole and projects onto the image plane behind, without being distorted. This hole is called the center of projection(COP). This is shown in figure 2.1. Of course, no real cameras have an infinitely small aperture. Real cameras use lenses to increase the light received. However, as long as an object is within the depth of field, or "in focus", its projection can be described by the pinhole model. A real lens will introduce some distortion, which the pinhole model does not consider. These distortions can be accounted for, but for the purpose of this section it is not important.

Projection geometry is easiest to do using homogeneous coordinates. This a type of coordinate that adds an extra element and is independent of scale. A homogeneous coordinate carries information only about the relative differences between its components. This is useful because the coordinates of a projected 3D point onto a plane is independent of the distance along the ray of projection. Linear transformations of the homogeneous coordinate can then be carried out without worrying about the scale. The coordinate can then later be converted back to real coordinates by dividing it such that its final element is equal to some defined value, usually 1.

The homogeneous coordinates of a projected point can be calculated linearly by a projection matrix P. This gives the homogeneous coordinates of the image point.

$$q = \boldsymbol{P}Q \tag{2.1}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \boldsymbol{P} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.2}$$

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.3}$$

Here q is the 2D homogeneous image coordinate and Q is the 3D homogeneous real world coordinate. Since q can be scaled by any constant and by definition still be the same coordinate, it essentially describes a line in 3D space. This is shown as the blue line in figure 2.1. If the distance to the image plane along the Z axis is 1, then by dividing q by a constant such that its third element is equal to 1, the point of intersection is found.

$$q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X\frac{1}{Z} \\ Y\frac{1}{Z} \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.4}$$

In reality, the image plane has a specific distance from the COP given in real life units like millimeters. This distance is called the focal length and varies

from camera to camera. The focal length, along with the pixel density are two camera intrinsic parameters that define what the the final pixel coordinates will be. The matrix transforming the image coordinates to pixel coordinates is called the intrinsic camera matrix. This matrix is essentially a rescaling and translation of the camera coordinate system.

$$\boldsymbol{K} = \begin{bmatrix} f\alpha_u & 0 & u_0 \\ 0 & -f\alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Where α_u and α_v are the pixel densities along the x and y direction, expressed in pixels per focal length. These rescale the X and Y axes. u_0 and v_0 are the coordinates of the image center, given in pixel coordinates. These move the origin to the top left of the image. The bottom right element, 1, is the value we define the focal length to be in our new re-scaled coordinate system. This value can be chosen freely, but must be respected when converting back to real coordinates. The direction of the y axis is changed to point downwards, hence the sign before $f\alpha_v$. Multiplying by q gives:

$$\boldsymbol{K}q = \boldsymbol{K} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Xf\alpha_u + Zu_0 \\ -Yf\alpha_v + Zv_0 \\ Z \end{bmatrix} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.6}$$

$[u, v]$ is then the final pixel coordinate. The result is scaled such that the third element is 1. This is because that is the defined distance to the image plane along the Z axis as defined by the intrinsic matrix.

This shows how the projection of 3D point can be calculated if the camera intrinsic parameters are known. It shows how any 3D point along a line through the COP will project to the same pixel values, creating a depth ambiguity. This is fundamental to all computer vision tasks and the fundamental concepts will be relevant for how poses are estimated from correspondences.

Chapter 3

Pose Estimation

3.1 Pose estimation introduction

The pose estimation problem is the problem of finding the six degree of freedom (6DOF) orientation and position of a rigid object from an image. This is a central problem in robotics, automation and computer vision in general. In these contexts it is often the 6D pose of an object with respect to the sensor that is of interest. That is, the rotation and translation of the object given with respect to the sensor coordinate system. With this information a robot manipulator could for example correctly orient a gripper to pick up and object.

Often the sensor consists of an RGB or RGB-D camera, where the latter also measures the depth information in the image with a separate depth sensor. Generally the RGB case is harder because of the inherit 3D to 2D projection ambiguity; all 3D points on a line through the center of projection will project to the same point in the image plane. The depth information is therefore lost and can be difficult to accurately recover. Because of this the RGB-D methods give more accurate results in general.

The pose estimation task can be, and has been, approached in many different ways. Traditionally, most single view RGB methods relied on local gradient information to detect features in the image. These found 2D to 3D correspondences using local invariant features such as [5, 1, 20]. Some also directly estimate the pose using template matching[11]. By using the object under different poses as templates, they search for matching areas in the image. A problem with these methods is that they rely on the object containing detectable features, which textureless objects have few of.

With deep learning it has become easier to deal with textureless objects. The deep learning networks can recognize whole objects, not just local patches, and can therefore more robustly find correspondences. All modern methods to some degree rely on deep learning. Mainly there are two approaches within deep learning pose estimation.

Those in the first category teach a model to find a set of correspondences be-

tween the image and the 3D object, and try to find the rotation and translation that best fits the set [9, 12]. With 2D-3D correspondences this pose is found by solving the Perspective-N-Point(PnP) problem. These methods mostly differ in how they establish these correspondences.

Those in the second category train an end-to-end model[18]. The end-to-end models are trained to directly find a pose from the input image. Some of these discretize the pose space and formulate it as a classification problem. The accuracy of these methods are limited because of the discretization, but the resulting pose can be refined with methods such as [16]. Others regress the pose instead, meaning the model learns to map images to poses.

Different methods are applicable in different use cases and data availability. For example, a method may provide high speed pose estimation, but fail under moderate occlusion. This could make it good for real time systems in a highly controlled environment, such as a production line, but unusable for autonomous driving in the real world, where robustness is critical. A method relying heavily on accurate depth information can be accurate at close range, but inaccurate at longer distances where the depth sensor is less accurate. Because of the different use scenarios there is always need to explore different approaches.

3.2 The main problems in pose estimation

There are some general problems in pose estimation from images that any method must overcome. The main problems come from visual ambiguities. Visual ambiguity is an umbrella term for anything that reduces the clarity of the image and may confuse a model. This can be lighting conditions, occlusion, symmetries, cluttering, multiple instances of an object, lack of textures and so on. While deep learning has improved upon the traditional image gradient methods, especially on textureless objects, there is still a problem with symmetries.

3.3 The problem with symmetries

Consider a model that detects corners and maps them to their corresponding corners on a 3D object. Then consider an object that is a mathematically perfect, textureless cube as in figure 3.1. For such a cube there is no way of knowing which of the eight corners is which. It is also not interesting to know which of the eight corners is which since it is of no consequence. It would be unfortunate for the model to "guess" at which corner it is, since it is both fundamentally impossible and of no interest. If seven corners are visible as shown in figure 3.1 and the model picked a random corner on the 3D object for each of them, there would be little coherence between each guess. The model could for example map all seven corners to the same corner on the 3D object. It makes more sense for the model to instead represent the correspondence of a corner as an equal probability for each of the eight corners. The model then

Source: https://www.seekpng.com/ipng/u2a9o0u2a9w7q8i1_blue-cube-png/

Figure 3.1: A perfect and textureless cube.

indicates that the eight possible poses are equally likely, which is correct.

Even if one is fine with the model simply guessing at which of the eight poses are correct, there would be a fundamental problem with training such a model. In most datasets, even symmetric objects have just one ground truth pose. This is the case for the datasets SurfEmb is trained on[2]. It then becomes challenging to define a loss function for the model. In the perfect cube example there can be eight identical input images that each have a different ground truth. This means the model cannot converge as its loss function is, from its perspective, inconsistent.

In reality there are of course no perfect symmetries, meaning it is theoretically possible for a model to differentiate them. In that case, there is always just one solution. However the model could overfit by learning to rely on minuscule differences in the corners. This makes the model more susceptible to noise, poor lighting and other visual ambiguities.

Because of this, many methods choose to handle symmetries explicitly. That is, their models do not learn the symmetries but are instead corrected and guided manually. For example [18] trains their model on rotations in a restricted range, such that the model only sees the object under the same symmetric mode. Then in order to find rotations outside this restricted range, they use a separate classifier that can give a rough estimate on the range from a holistic look at the object. Their restricted model can then be used to find any pose by combining the two.

While explicit solutions can handle global symmetries it requires some setup work and does not generalize between objects. It does also not handle other kinds of visual ambiguities such as occlusion and lighting. These ambiguities can have the same effect as symmetries; they increase the number of possible correspondences. As mentioned it makes more sense to have a distribution

over the possible matches rather than just a single guess. This gives more information about the how sure the model is and what other object points are possible matches.

SurfEmb attempts to solve the problems with ambiguities by introducing a method for their model to implicitly learn the ambiguities of an object, most notably symmetry. Their model can give multimodal distributions that can indicate that a pixel correlates with multiple object points. In the cube example this would in theory mean a pixel displaying a corner will correlate equally to all eight corners. How this is done will be discussed more in the section about SurfEmb.

After the correspondences are established, SurfEmb, like many other methods, solve PnP problems in order to find pose estimates.

3.4 The PnP problem

The PnP problem is a fundamental problem in computer vision. It is the problem of fitting a set of 2D-to-3D correspondences such that the 3D points project onto the 2D points from the cameras perspective as shown in figure 3.2. This problem arises when points on a 3D object are identified in an image. Because of the projection ambiguity as discussed in the camera theory section, the 3D object point displayed by a pixel may lie anywhere on a line through the COP. The "n" in PnP refers to the number of corresponding points and lines. n=3 is the smallest problem size with a finite number of solutions, yielding up to four solutions.

Many different methods for solving the non-minimal PnP problem have been presented, and there is no consensus that the "optimal solution" has been found. Some popular methods include EPnP[15] and [24]. EPnP was published in 2009 and achieved linear run-time with respect to n, which was a leap from the previous state of the art methods with runtimes of $n^5 - n^8$, according to the authors. EPnP and other methods rely on a clever reformulation of the problem. [24] for example reformulate the PnP problem into an optimization by parmameterizes it with non-unit quaternions.

Since it is a minimal problem, the P3P problem can be solved using geometric constraints. The method used by SurfEmb is one dedicated only to the P3P problem[13], published in 2017. They find an efficient algebraic solution to the trigonometric equations, avoiding unneccessary and unstable intermediate results that previous methods included. The math of the PnP problems can become rather complicated and so will not be covered.

If all correspondences are assumed to be correct, and there is no noise in the measurements, then one of the solutions to the minimal P3P problem will be the correct pose. Determining which is the correct one may not always be as simple if no other data is available to resolve the ambiguities. If $n \geq 6$, there will always be a unique solution. Additionally, in the real world there is noise in the measurement, so a greater number of points will reduce the effect of random noise. It is therefore beneficial to have many correct correspondences.

If all correspondences are correct, and there is random noise in the measurements, the solution should be the pose with the least error. The mathematical solution should find the pose with the smallest sum of squared distances from the projection of the 3D points to their corresponding image points. Referring to figure 3.2, this means the distance in the image plane from an image point x_i to its projected correspondence X_i. This distance is called the *reprojection error*.

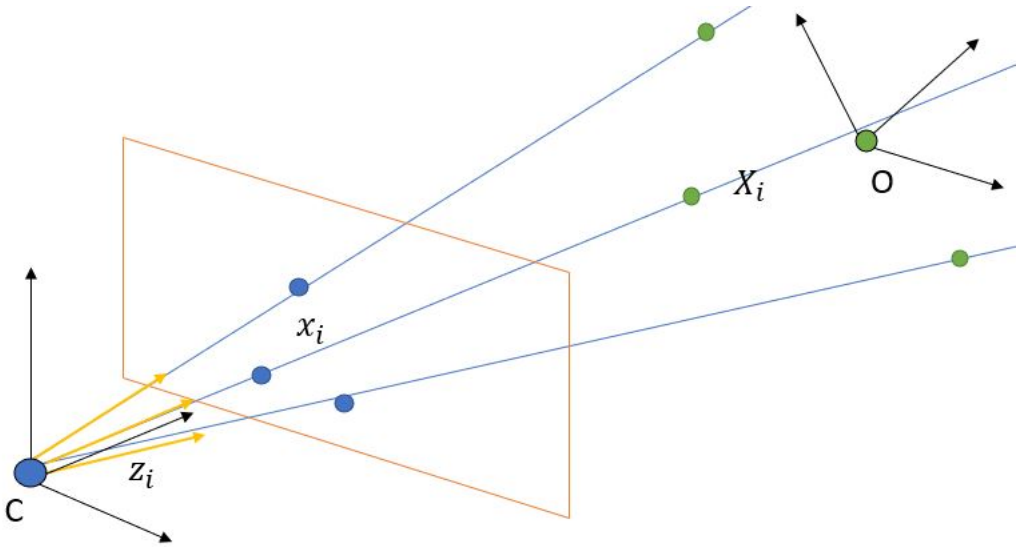


Figure 3.2: 2D to 3D correspondences. The camera's center of projection is marked C, the objects coordinate system is marked O. The blue points are image points corresponding to the green object(3D) points. The blue lines show the possible positions of the object points and extend to infinity. The yellow vectors z_i are the unit vectors describing the direction of each line.

Mathematically, the least squares solution to the PnP problem can be expressed as follows:

$$\boldsymbol{R}_{\boldsymbol{o}}^{\boldsymbol{c}}, \boldsymbol{t}_{\boldsymbol{o}}^{\boldsymbol{c}} = \text{argmin } J \tag{3.1}$$

$$J = \sum_i ||z_i - \frac{1}{\alpha_i}(\boldsymbol{R}_{\boldsymbol{o}}^{\boldsymbol{c}} X_i^o + \boldsymbol{t}_{\boldsymbol{o}}^{\boldsymbol{c}})||^2 \tag{3.2}$$

$$\alpha_i = ||\boldsymbol{R}_{\boldsymbol{o}}^{\boldsymbol{c}} X_i^o + \boldsymbol{t}_{\boldsymbol{o}}^{\boldsymbol{c}}|| \tag{3.3}$$

$$\text{Subject to: } \boldsymbol{R}_{\boldsymbol{o}}^{\boldsymbol{c}} \in SO(3) \tag{3.4}$$

Where $\boldsymbol{R}_{\boldsymbol{o}}^{\boldsymbol{c}}$ and $\boldsymbol{t}_{\boldsymbol{o}}^{\boldsymbol{c}}$ are the rotation matrix and translation vector of the object given in the camera coordinate system. X_i^o are the object points given in the object frame, and z_i are the unit direction vectors of the correspondence lines. α_i is the distance of each object point to the camera under a given pose, and is

used in the cost function J to calculate the unit vector describing the direction of the line through object point X_i. Technically, the cost function in this case does not calculate the reprojection error, but instead the distance between the unit direction vectors visualized by the yellow vectors in figure 3.2. This measurement generalizes better towards the edges of the image where distances in the image plane are amplified, especially noticeable with a high field of view.

Until now all correspondences have been assumed to be correct, with only measurement noise. The non-minimal PnP solvers EPnP[15] and [24] rely on this assumption. Unfortunately, correspondences are computed by an imperfect algorithm or network and are therefore not always matched correctly. Ambiguities such as lighting, symmetries, occlusion and cluttering will affect the matching method negatively. Correspondences are often labeled as either inliers or outliers. An inlier is defined as a correspondence within a set error threshold, whereas an outlier is outside and should be ignored. If outliers are included when the least squares solution is calculated it will severely impact the accuracy of the solution. It is therefore critical to identify these outliers and remove them. Several methods exist for this purpose, one well known and widely used being RANSAC[7], presented in 1981, which SurfEmb uses a simpler version of. In a RANSAC scheme, a minimal P3P solver can help identify outliers.

3.5 RANSAC

RANSAC takes as input a dataset and a mathematical model with unknown parameters that will be estimated to best fit the inliers of the dataset. The method is iterative and has five steps which are repeated:

- The method selects a random sample consisting of the minimal number of datapoints from which the model parameters can be extracted. For a linear model this would be two datapoints, since a line can be defined by two points. These points are for now assumed to be inliers.
- Model parameters are fitted to the sample.
- The model is applied to the whole dataset.
- Depending on a model specific loss function, some datapoints that fit the model well enough are determined to be a part of a consensus set. For a linear model the loss function would be the distance from each point to the line.
- The model is then improved by fitting it to the whole consensus set plus the original sample. Often a least squares solution is used, such as in the re-projection error of PnP or the regression of a line.
- The improved model is tested on the entire dataset, its total loss is compared to the previously best model. If it is better, then it becomes the new best model.

What makes RANSAC simple is also what can make it slow. The RANSAC method is non-deterministic since it is inherently random. This means it is not possible to know how many iterations are needed for a given model accuracy, or how good a model will be after a fixed amount of iterations. The method relies on randomly selecting a good minimal sample which can get very unlikely when many points are needed in the minimal solution, especially if there is a high outlier fraction. Therefore its performance deteriorates quickly with higher outlier fractions and with higher number of model parameters. [23] showed that RANSAC gets outperformed by newer methods such as Graduated Non-Convexity(GNC) on point cloud registration. While it is able to achieve similar accuracy, its runtime increases exponentially with respect to the number of outliers, whereas GNC runtime remains constant. In their example RANSAC had 10 times the runtime of GNC for 80% outlier fraction for the same accuracy. An advantage of RANSAC is that its scheme can easily be used in many different parameter fitting problems, whereas GNC is mostly specialized towards point-cloud fitting.

When using RANSAC in the PnP problem, the dataset is the set of correspondences. The minimum sample size will be 3 correspondences and the model by which the dataset should be explained is the projection of the rotated and translated 3D object points. The loss function can be the square of the reprojection error explained previously. This application of RANSAC will be referred to as PnP-RANSAC.

SurfEmb does not use this exact PnP-RANSAC scheme. Why this exact scheme cannot be used is due to their correspondences consisting of distributions rather than one-to-one matches. Their PnP-RANSAC method will be explained more in the SurfEmb section.

To summarize, by using RANSAC and a P3P solver it is possible to recover the 6DOF pose of an object given a set of correspondences that may contain outliers. Before this can be done the correspondences must somehow be found. Like most modern methods, SurfEmb uses deep learning to find these correspondences. The next two sections are dedicated to explaining how this is achieved.

Chapter 4

Machine Learning

4.1 What is machine learning?

Previously, any model explaining a dataset had to be determined by humans. Johannes Kepler(1571-1630) was a German mathematician and astronomer who attempted to explain the movements of planets. The data he based his model on was collected by his mentor from naked eye observations. This was before Newton was born, so Kepler did not have Newtons law of gravity at his disposal. Kepler chose the simplest geometric model he believed could fit the dataset, which based on observations was elliptical orbits around the sun. In order to accurately explain their movements, he needed to know the eccentricity and size of the ellipse. Kepler split his dataset and used one for finding the parameters of his model, and one for validation. He started with a guess of the eccentricity and size of the ellipse, and iteratively adjusted them until the model fit the data better. Finally he validated the model on the separate validation set. What Kepler did was essentially machine learning without a machine(pg. 103-105 in [22]). Today, machines can very efficiently update the model parameters and fit much more complicated models.

With Keplers method in mind, machine learning can be summed up as such:

- Have a dataset from which to build a model. Split it into a training set, a validation set and a test set.
- Find the simplest model that can fit the data. For Kepler this was an educated guess on a specific model.
- Define a loss function that tells the computer if the output is good or bad. Differentiate the loss function with respect to the model parameters.
- Run the training set through the model and calculate the loss. Back propagate the derivatives of the loss to find the direction minimizing it in parameter space.

- Have the computer update the parameters with a step size equal to some constant times their respective derivative. This constant is called the learning rate.
- Continuously check the model against the validation set to see if the model performs on data it has not been trained on directly. The model should be adjusted such that the validation loss also decreases, to ensure that the model is able to generalize.
- Finally, when the model is deemed to perform well enough on the validation set, the model can be tested on the test set to see if the model performs on unseen data.

The second step assumes that a model with unknown parameters is known or found beforehand. It is often difficult to make an educated guess of what the best model is, and it may be infeasible for complicated datasets. Even Kepler spent years trying to make sense of the observations he had. The model itself is sometimes the unknown, not just the parameters. Lets say one would like to make a model predicting the stock market based on all kinds of demographic and economic data. It is a highly complex system that cannot be explained by for example physics. Attempting to find a model quickly becomes extremely difficult as the number of inputs increases.

The solution is to exploit the high computational power of modern computers by making a very flexible model with many parameters, inspired by how neurons work in the brain. These are called neural networks, simple examples of which are shown in figure 4.1. Nodes are connected to each other in layers. Each node gets some input values from previous nodes, perform a transformation, and give an output to some nodes in the next layer. Perhaps the most simple neural network is the Multilayer Perceptro(MLP) which is a fully connected network consisting of three or more layers of nodes. A network is fully connected if each node is connected to all nodes in adjacent layers.

Large neural networks can have millions or even billions of parameters and are therefore able to morph in countless ways and achieve high accuracy. Millions of parameters would be a bit much for Kepler to deal with, but a sufficient computer can easily process them in milliseconds. This does give us less control over what the final model looks like, but the complexity of the model can still be adjusted by choosing an appropriate number of parameters.

The task of finding a model instead becomes the task of choosing the right network architecture, which in most cases is a lot easier. The disadvantage with such flexible models is that they require a lot of training data, the absence of which is a limiting factor in many cases. With enough training data the network is able to learn complex relations. How this is achieved and the mechanics of learning will be discussed next.

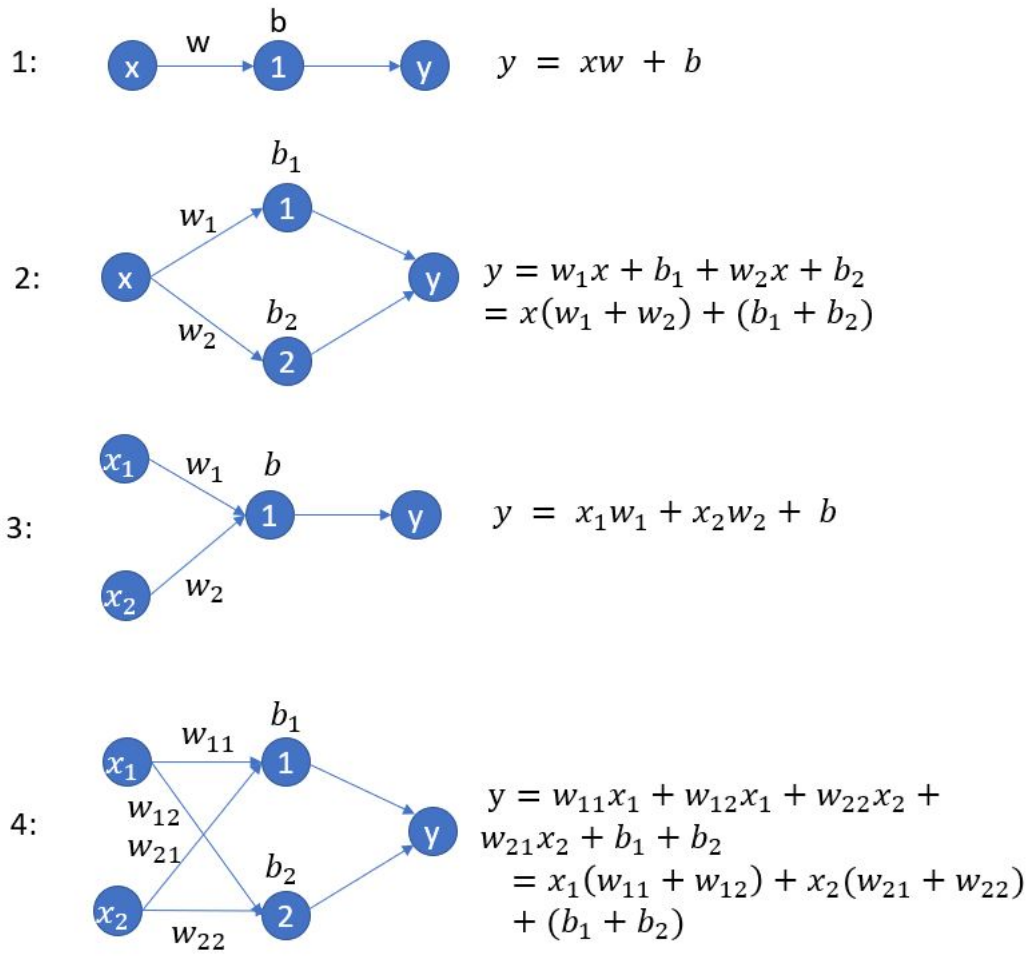


Figure 4.1: The simplest possible neural networks of the Multilayer Perceptron class, in which nodes of a layer are connected to all nodes in their adjacent layers. While they vary in numbers of inputs and parameters, the output y is always linear with respect to the input values x. This can be seen in the expressions for y, which are essentially the same for models 1 and 2, and for models 3 and 4.

4.2 Achieving complexity

In figure 4.1, the input values at each node are only scaled by a weight and added to a bias. This is the definition of a linear transformation. Linearity has the property that for two linear transformations $f(.)$ and $g(.)$, $f(g(.))$ and $g(f(.))$ are also a linear transformations. This means no that matter how many layers or neurons we add, the model will still be linear.

This is insufficient for modeling anything that is non-linear. In order to introduce non-linearity to the model, activation functions are added. At each node, the input value is still scaled by a weight and has a bias added, but the result is then used to evaluate an activation function, the resulting value is then outputted. Examples of these functions are shown in figure 4.2. By chaining many of these activation functions, more complex functions can be formed. A neuron with an activation function, and a single input value does the following calculation:

$$o = act(wx + b) \tag{4.1}$$

Where $act(.)$ is the activation function, o is the output value, x is the input and w and b are the learnt parameters of the node. If the neuron has multiple inputs, then w and x become vectors with dimension equal to the number of inputs.

4.3 The mechanics of learning

4.3.1 The loss function

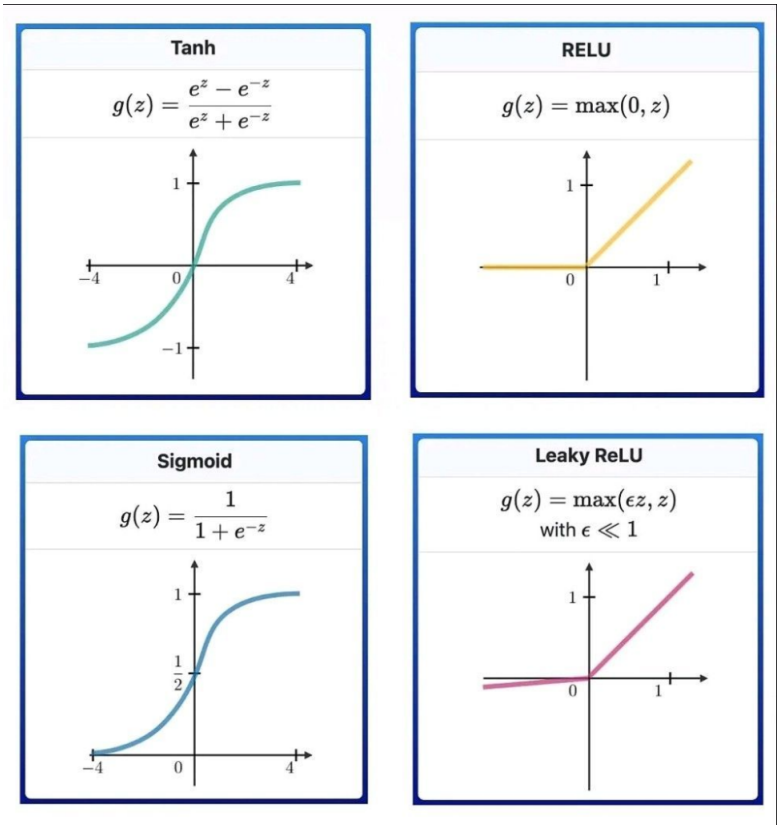
With the activation functions the network is able to give complex non-linear outputs. Now the network needs to know how it should adjust its parameters. This is defined by the loss function. The loss function takes the output of the network and performs some transformation, often comparing the output with the desired output associated with the given input. The loss function is defined such that a low loss is good, so the goal of the network is to minimize this loss function.

4.3.2 The backwards pass

In order to minimize the loss function, the network needs to learn the desired relationship between its input and the loss function. For a model to learn its parameters, it must know how each one affected the outcome, i.e. the loss. This is the purpose of the backwards pass. It is so-called because of the way the partial derivatives are calculated from the output and backwards to the input, using the chain rule for differentiation. The loss function is what we want to minimize, and is dependent on the last layer of the network, which in turn is dependent on the previous layer, so on and so forth. The cost of the backwards pass increases with the number of parameters. The backwards pass gives the gradient of the loss function in parameter space, which points in the direction of largest increase of the loss.

4.3.3 Learning rate

The network the needs to move some distance in parameter space in the opposite direction of the gradient. The adjustment of each parameter is proportional to their derivative. A constant is multiplied by the derivative called the learning



Source: https://ai-artificial-intelligence.webyes.com.br/most-used-activation-functions-in-neural-networks/

Figure 4.2: The most common activation functions. z refers to the sum of all input values of the node.

rate. Like in an optimization task using only gradient information it is impossible to know for how long the loss function decreases in that direction. If the network moves too far then it might move away from a global minimum and instead increase the loss. Therefore the learning rate should be small enough for this to not happen too much, but also large enough for the network to learn efficiently. The learning rate is a hyperparameter that can be adjusted based on testing.

It should also be mentioned that most networks are trained with stochastic gradient descent. A "vanilla" gradient descent calculates the gradient after processing all inputs in a training set. The total loss on the training set is what should be minimized, so this may seem like a good way to do so. The problem with that approach is that the network can become stuck at local minima. By

introducing randomness to the gradient direction the network can overcome local minima. The stochastic variant introduces randomness by taking in small batches of random inputs and calculate the gradient for that batch only. The network then updates its parameters between each batch([22] pg.184). Since the gradient is partly random, the learning rate needs to be small enough not to diverge too far off the "vanilla" gradient.

Using the theory up until now one can make an MLP and train it on some data. Unfortunately the MLP has some problems when it comes to large scale inputs such as images.

4.4 Images as input

A problem with fully connected networks is that the number of parameters increases quadratically with the layer size. Since every node in a layer is connected to every node in the next layer, it increases as $n*m$ for each pair of layers, where n and m are the number of nodes in the respective layers. When the input has a high dimension then the layers must also be larger to interpret the information.

An example of inputs with high dimension are images. Consider a 300x300 color image. The image has 90.000 pixels, each with 3 channels, giving an input size of 270.000. Features in the image need to be detected in order to understand it. If the first layer has 90.000 nodes, each with a weight and bias, then the number of parameters is $270.000 * 90.000 * 2 = 48.600.000.000$. If represented by 32bit float numbers then the memory requirement for this first layer is 194 gigabyte. Computing the gradient with respect to each of these parameters also becomes very costly. This puts a limit at how deep a fully connected network can be. Having a deep network is important since the complexity of the model can only increase with the number of parameters and activation functions chained together.

Luckily the information in images can be extracted more efficiently. Using a fully connected network to interpret images is inefficient since the basis of a fully connected network is that there is some meaningful non-zero connection between each node that should be adjusted. The value of a pixel in the far left of the image seldom influences the value of a pixel in the right side of the image. There is therefore little point in a node taking inputs from both of these pixels and attempting to adjust their weights in some meaningful way.

Not only are fully connected networks incredibly inefficient at image interpretation in terms of the number of parameters, but they also hinder the network in being translation invariant. Consider an image with a plane in the top left corner. A fully connected network can only learn to identify it as a plane if much of its training data contains planes in the top left corner. If its training data does not contain planes in that exact location the network does not recognize it, as the weights and biases from those pixels never learned to identify planes. To a human it seems natural that a plane is a plane no matter where in the image it is, but to a fully connected network it is a whole different input. Ideally a network attempting to understand images and recognizing objects should be

translation invariant.

Another type of network is needed to solve these problems that occur with having images as input. A Convolutional Neural Network(CNN) is such a network architecture that specializes in image interpretation.

Chapter 5

Convolutional Neural Networks

Previously, functions, called descriptors, were handcrafted by engineers to pick out certain features[1, 5]. A descriptor usually uses the gradient information of a local patch of pixels to produce an output value indicating the presence of some feature. With the introduction of Convolutional Neural Networks, this method became largely obsoloete. A CNN learns its own descriptors, called filters, which can be chained together to get a holistic understanding of the image. This turns out to be both easier for the engineers and better performing.

A Convolutional Neural Network is so called because it incorporates convolution to extract features in the input. A feature might be a corner, an edge or some other shape. It can also be a higher level feature that humans would not consider a shape, but the network has learned as a smart way of recognizing something of interest. It also learns hierarchies of patterns by chaining convolution layers. A first layer can detect basic shapes such as edges and corners, and then a second layer can detect larger shapes such as squares or circles as combinations of the edges and corners. This makes it good for extracting the important contents of images.

Mainly there are two ways in which a CNN solves the previously discussed problems with using an MLP for image interpretation. Firstly, this type of network solves the problem with too many parameters by having each node only consider a certain area, called its *perceptive field*. The perceptive fields of neurons are indicated in figure 5.3 as the small squares. As discussed this makes sense for image interpretation as pixels are mostly dependent on their nearby pixels. Whereas a fully connected network learns global patterns involving all pixels, a CNN learns local patterns([3] pg. 204-205).

Secondly, it achieves translation invariance by using the same set of parameters for each node in its convolution layers. Therefore it does not matter where in the image the feature is, if it has been learned to be detected it can be detected anywhere([3] pg.204-205). These sets of parameters are what make up

the filters used in convolution to detect features. As the same filters/parameters are used by each node it further reduces the number of parameters that need to be stored. Filters and CNN structure will be discussed further. First, one must understand why and how convolution is used to detect patterns.

5.1 Convolution

A convolution is an operation taking as input two functions and combining them in a specific way to produce an output function. The output function contains useful information about the two input functions. Mathematically it is expressed as follows for two continuous one dimensional functions:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{5.1}$$

For most real world applications, the two functions do not go from $-\infty$ to ∞. For such functions with finite domain, one may think of them as being zero outside of their domain.

Starting in opposite ends, the two functions are multiplied together and the product is integrated. For functions with finite domain it is useful to picture it as a sliding of one function over the other as shown in figure 5.1. For this visualization, the latter function $g(x)$ must first be flipped about the y axis, i.e. $g(-x)$, moved left until there is no overlap and then slid over f from the left. The resulting integral, or convolution, is a function of t, where t is the position of $g(-x)$ relative to its original position. As t increases, g is slid accordingly further over f. In the overlapping regions, the product of the two functions is integrated.

Convolution is useful in many contexts. The above definition is however not so useful in real world contexts, as it is concerns only continuous functions extending from $-\infty$ to ∞. In the real world one often wants to measure similarity between discrete functions that have a finite domain, as is the case in for example signal processing. This is also what is needed for CNNs. For this purpose one must instead use discrete convolution. In CNN contexts the flipped function $g(-x)$ is called a *kernel* when one- or two-dimensional, or *filter* when three-dimensional. The discrete convolution for two one dimensional functions then takes the following form:

$$(f * g)[t] = \sum_{n=-\infty}^{\infty} f[n] * g[t-n] \tag{5.2}$$

$$k[t] = g[-t] \tag{5.3}$$

Where $k[t]$ is the kernel. Equation 5.4 shows how such a convolution is performed. By looking at the 6 resulting equations one can see that it is as though the kernel is slid from the left and multiplied by the two currently overlapping values of f.

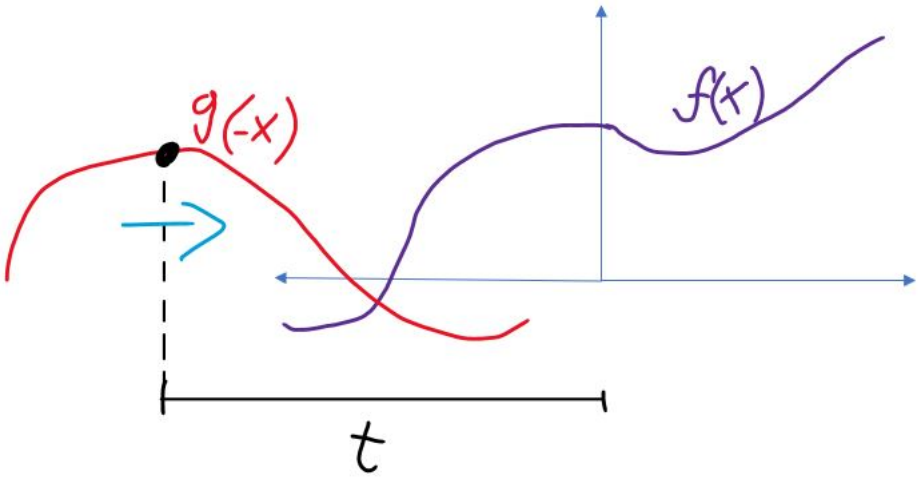


Figure 5.1: An intuitive visualization of convolution. The function g is flipped and then slid over f from the left. The black dot indicates the location where g originally intersected the y axis. In this example the two functions have a finite domain. The only part of the overlap that adds to the final integral value is the overlap.

5.2 Finding patterns with convolution

As discussed, the goal with using convolutions in image recognition is to extract features such as corners, circles, shapes in general, but also higher level features that the model learns on its own.

This section focuses on the method for detecting traditional shapes such as edges and corners. The outline of shapes can be identified by looking at the image gradients, as they will have large values if the shape is contrasted against the background. Convolutions can be used as measurements of image gradients if the kernel is chosen appropriately. A convolution does not generally represent similarity between the shapes of f and k. However, this can be achieved if the kernel values are centered around 0. That is, if one wants to check for a decrease in f, the kernel should go from positive to negative. With such a kernel, the convolution becomes a measure of similar trends, not of the actual values. The following equations give an example of this for a kernel detecting a negative gradient:

$$
\begin{aligned}
f &= \{10, 8, 6, 2, 0\} \\
k &= \{1, -1\} \\
\Rightarrow g &= \{-1, 1\} \\
1 &: -1 * 10 = -10 \\
2 &: -1 * 8 + 1 * 10 = 2 \\
3 &: -1 * 6 + 1 * 8 = 2 \\
4 &: -1 * 2 + 1 * 6 = 4 \\
5 &: -1 * 0 + 1 * 2 = 2 \\
6 &: 1 * 0 = 0 \\
\Rightarrow (f * g) &= \{-10, 2, 2, 4, 2, 0\}
\end{aligned} \tag{5.4}
$$

With the kernel $k = \{1, -1\}$ the scores give the drops in value between each element in f. The value t maximizing $(f * g)[t]$ will be the window where the shape of f correlates the most with k. In this case the largest value is given to the largest drop.

If the kernel has the same trends, but with strictly positive numbers:

$$
\begin{aligned}
k &= \{3, 1\} \\
\Rightarrow g &= \{1, 3\} \\
\Rightarrow (f * g) &= \{10, 38, 30, 20, 15, 27\}
\end{aligned} \tag{5.5}
$$

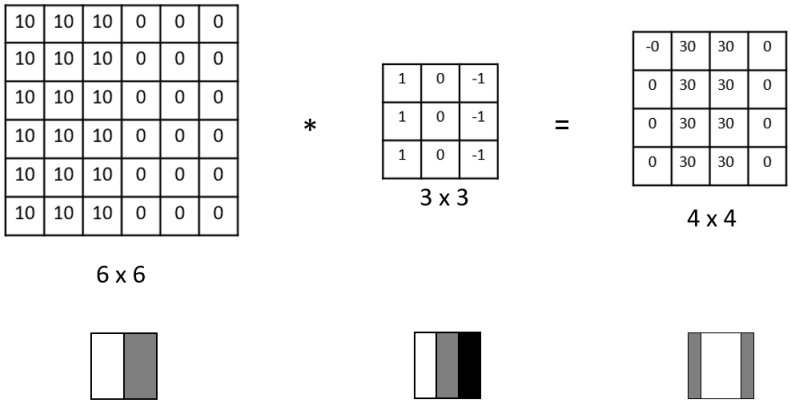
The highest score is given to the drop from 10 to 8, since the kernels values are all positive and is therefore biased towards positive numbers. This kernel is therefore not suited for detecting gradients.

In the case of single channel images like for grey-scale images, the convolution is two dimensional. For images with multiple channels like colored images, the convolution is three dimensional and the kernel is referred to as a filter.

The point of this section was to show how convolution with kernel values around 0 can detect shapes. While the detection of traditional shapes is critical, this does not mean that the network only learns to use such kernels. One of the benefits of CNNs is that they can learn to use kernels that humans have not thought to use

5.3 Convolution of images

As shown, a kernel appropriately centered around 0 can be used in a convolution to detect shapes in a function f. An example of a two dimensional convolution of an image utilizing this for edge detection is shown in figure 5.2.



Source: https://stats.stackexchange.com/questions/535456/edge-detection-convolution-intuition

Figure 5.2: From left to right, an input image is convoluted with a 3x3 edge detection kernel, creating a 4x4 output. The kernel shifts over the whole image evaluating each 3x3 neighbourhood and outputting the resulting number to the corresponding index in the output layer. The decrease in dimension from 6x6 to 4x4 is due to the fact that a 3x3 kernel cannot shift more than three times in a direction without going outside the image.

5.4 CNN architecture

For each filter/kernel used, a new image is formed, called a feature map. Usually more than one filter is used, meaning the original image gets multiplied. Each of these new feature maps is called a *channel*. In order to introduce non-linearity an activation function is applied to each of the elements in the feature maps. ReLU is a popular choice for this purpose, shown in figure 4.2.

Lets say one wants to train a network to count the number of circles in an image using these edge detection kernels. By using multiple such kernels oriented in different directions, the image gradients in different directions are sampled. Some sections in the image will have a detection of edges in all directions, indicating a likely circle. The next convolutional layer can then use a circle detection kernel. However, there is a problem. The next convolutional layer would need a kernel the size of the original circle in order to incorporate all the detected edges. For detecting large circles one then needs an equally large kernel. The solution to this is to decrease the dimensions by down-sampling between convolutions. By down-sampling multiple times, circles of all sizes can eventually be detected. As subsampling decreases the dimension of each feature map, more kernels are usually used, increasing the number of channels. This is shown in figure 5.3.

Looking at figure 5.3, the feature maps decrease in size between each convolution layer. In the figure this is indicated by the "Subsampling" arrows. This is a general term as there are multiple ways of subsampling a feature map. Two common ways of reducing the dimension of the image is strided convolution and pooling.

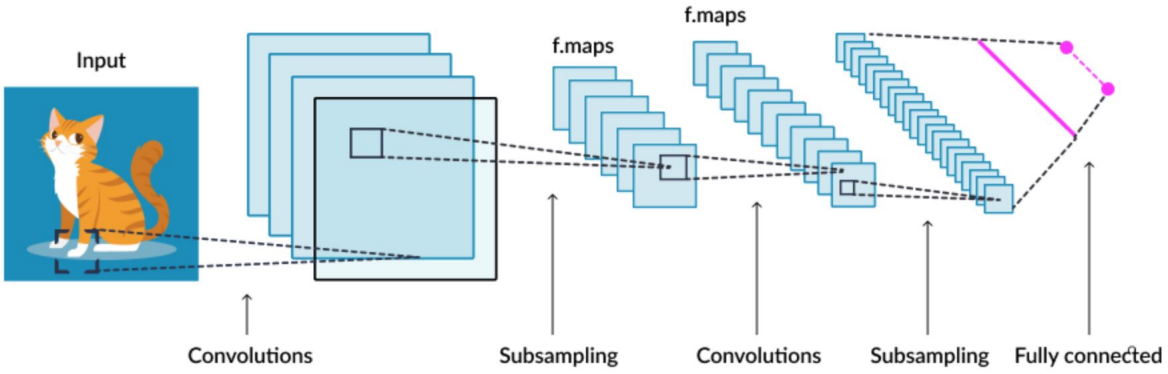
A strided convolution is a convolution where the kernel moves more than one element each time it shifts over the image. The number of elements the kernel moves in-between each dot product is called the stride. The way a discrete convolution is defined, the kernel moves one element at a time, so a stride of one. If we instead select a bigger stride, this has the effect of reducing the dimension of the output since there are fewer evaluations. Like with any convolution, the model can learn the kernel parameters for a strided convolution.

Pooling is very similar. It also uses a window of a fixed size that moves with a stride greater than one. Instead of taking the dot product between the kernel and matrix, it instead selects one element as its output. For max-pooling this would be the element with the highest value. While the parameters of a strided convolution can be learned, pooling layers are fixed by the programmer. This means the network cannot improve upon them, but it also gives less computing cost. Back-propagating derivatives is faster through a max-pool layer, since the derivatives of the elements that were not selected is simply zero as they did not affect the outcome. This cuts off the "partial derivative chain" all the way back to the original input, saving a lot of computing in the backwards pass. Whereas in a strided convolution, each element in the original matrix may retain a derivative.

After the features have been extracted and the dimension has been reduced, the final layers is often a fully connected neural network. This is to reduce the high level features of the final feature map to a useful output. In a classification case it would be a vector containing the score for each class. In the circle counting example it would be a single integer.

5.5 ResNet, U-Net and ResUNet

A CNN is usually a very deep network, which is good for understanding complex images. There is however a problem with back-propagating such a deep network. [3] uses the analogy of the Game of Telephone, which is the game where one person whispers a message to another, which whisper it to another, and so on. With enough people the message will at the end be very different from the original message. The degradation of the message for each person is essentially the same as noise in a CNN between each layer. The introduction of noise comes from the many chained activation functions which each introduce some noise. This can be from functions such as the sigmoid function that saturate when values get too large or too small, causing the gradient to become very small. The gradients are multiplied together when back-propagating with the chain rule, and so multiple saturated activation functions will decrease the gradient exponentially.



Source: https://www.skyfilabs.com/project-ideas/image-classifier-for-identifying-cats-dogs

Figure 5.3: A typical CNN structure. The feature maps are down-sampled and mutiplied for each layer. In the end the network has some fully connected layers to reduce the output dimension.

ReLu is an activation function that never saturates for positive inputs, but has a constant zero for negative values, meaning the gradient is also zero for all negative values. Any parameter taken in by the node with a ReLu function outputting a zero is therefore ignored.

The solution is simple: take the input to such a layer and add it to its output. Then the network retains a noiseless channel through which it can back-propagate. This was introduced in 2015 by Microsoft, and the family of such networks were named ResNet[10].

SurfEmb uses a type of CNN called ResUNet[6], introduced in 2019. ResUNet is an architecture that combines a U-Net[19] architecture with residual connections like those in ResNet.

U-Net was developed for semantic segmentation. Its input is an image, and its output is an image colored based on the different segments of the image. A typical CNN used for classification will, like in figure 5.3, reduce dimensions until the final output is a vector. A U-Net must also compress the dimensions in order to understand the image, but must then somehow regain the dimensions in order to output a new image of the same size as the input. For this purpose the U-Net uses a series of transposed convolution layers to up-sample the feature maps, giving it its characteristic "U" shape like shown in figure 5.4. A transposed convolution is a type of convolution that up-samples the input, increasing dimension. It is therefore labeled as an encoder-decoder network. As seen in the figure, U-Net itself also uses feed-forward connections to help with up-sampling the encoded features. The authors state that they were introduced to help the network with up-scaling and localizing the position of the different

segments. They were not introduced to solve the vanishing gradient problem. In fact, U-Net was published around 8 months before ResNet. A residual connection does not skip as many layers and is introduced to solve the vanishing gradient problem by upholding a direct noise free path.

ResUNet improves on the U-Net architecture by replacing each of the building blocks with a residual block[6] containing an internal residual connection. Both the encoder and decoder part of the network then have multiple residual connections. The main benefit of this is that a large depth is possible and learning becomes easier. With the covered theory it is possible to get a basic

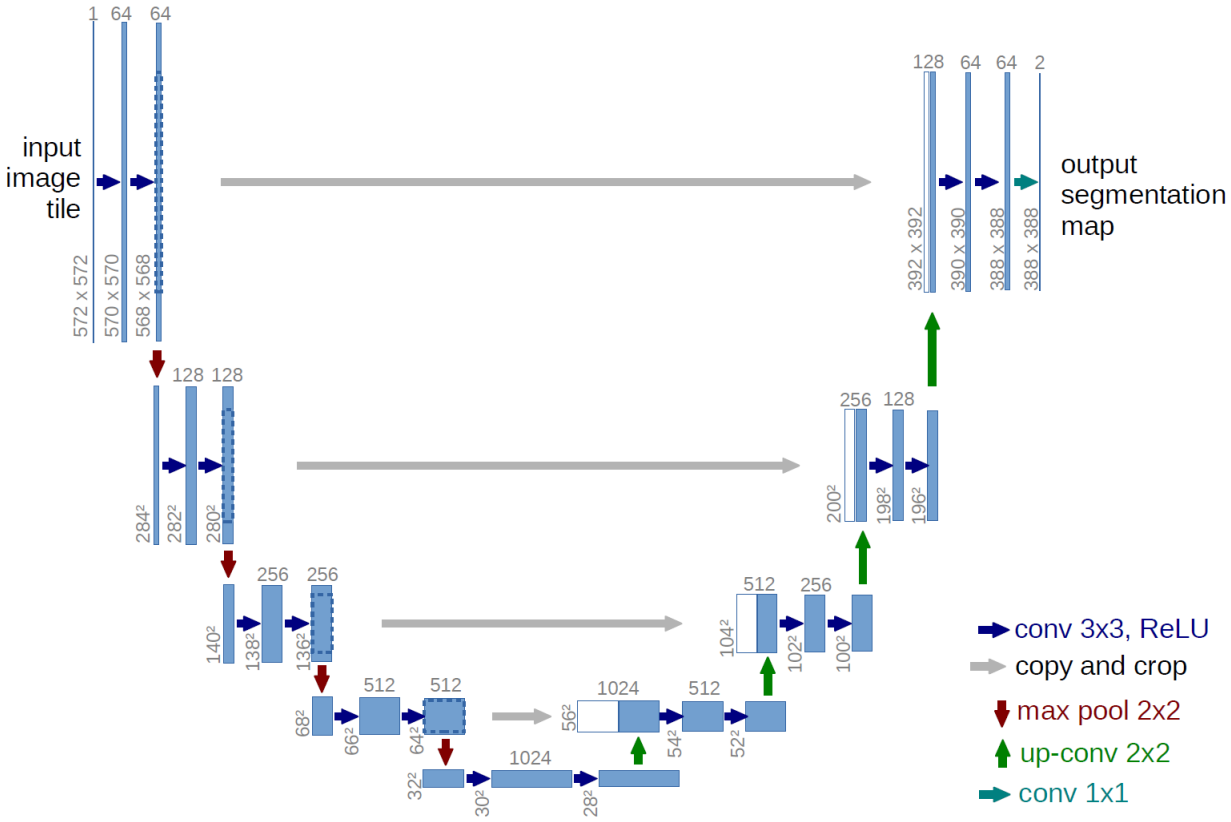


Figure 5.4: The U-Net architecture. Taken from [19]

understanding of how SurfEmb works.

Chapter 6

SurfEmb

SurfEmb is a 6DOF pose estimation method trained to find the poses of a set of 3D objects in a scene. It has one version that only uses RGB data and another that incorporates depth information aswell. The dataset on which it was trained on by the authors are the datasets found on the BOP database(https://bop.felk.cvut.cz/datasets/). The BOP website is a hub that facilitates competition between different pose estimation methods. The website provides datasets for training and scores the methods based on a hidden test set of unseen data. When SurfEmb was published in 2021 its RGBD version became state of the art. Due to the rapid developments in the field, in 2022 some methods have overtaken it, but SurfEmb's RGBD version still ranks as tenth place in overall score. The RGB and RGBD versions are the same, the only difference is that the RGBD version refines the final pose translation using depth information.

6.1 The basics

SurfEmb is based on finding dense correspondences between the image and the 3D object, and then using a modified PnP-RANSAC scheme to find a good pose estimate. The pose estimate is then refined by local optimization of the pose score function, which will be explained later. In order to locate the object in the image they use CosyPose's[14] maskRCNN model.

What makes SurfEmb special is the representation of correspondences. Where a traditional correspondence consists of one pixel and one object point, SurfEmb creates a probability distribution over all object points for a given pixel. The distribution can be expressed as follows:

$$p(c|I, u, u \in M) \tag{6.1}$$

Where I is an image, c is an object point, u is a pixel and M is the set of all pixels in the object mask. Such a distribution can handle symmetries as it can represent a correspondence to more than just one object point. In the perfect, textureless cube example, this would in theory mean each pixel on a corner

gets a 12.5% likelihood score on each of the eight corners. This is a multimodal distribution, and the authors claim that, to the best of their knowledge, SurfEmb is the first method to be able to represent such distributions.

Given a pose, a probability score for that pose can be calculated by looking at all the pixels in the image and how well each correlates to the object point it is displaying. The method then attempts to find the pose that maximizes the total probability score. For the textured, imperfect cube, the method will have some symmetrical and some unsymmetrical correspondences. The unsymmetrical correspondences will pull the solution space towards the unique correct solution, and the symmetrical correspondences will not cause problems since they agree just as much with any of their symmetries. Consequently, the unique, correct pose is in theory guaranteed to be the one with the highest probability score.

In order to find this pose SurfEmb uses a P3P solver in a scheme similar to RANSAC. They randomly choose sets of four correspondences, pixel-object point pairs, in a way where high probability pairs are likely to be picked. They choose 10.000 such sets before all of them are sent through a P3P solver to calculate 10.000 poses. Some poses can easily be pruned away as they are either very far away or one of the four object points used to calculate the pose is not visible under the pose. The surviving poses are sent through a scoring function that scores each based on the total probability, found by looking at how much each pixel corresponds to the object point it is displaying. The highest scoring pose is sent to further pose refinement by local maximization of the total probability. The pose estimation pipeline is shown in figure 6.3.

6.2 How multimodal distributions are achieved

To achieve the distribution over object points, the idea is to create two models. One query model to transform pixels in the image, and one key model to transform 3D object points sampled from the 3D object. These models embed the pixels and object points to a common embedded space. An embedded pixel is called a query, and an embedded object point is called a key. The level of correspondence between a pixel and an object point is then measured as a function of the dot product between their query and key. Approximately 75.000 object points are sampled from the object surface.

The query model has a ResUNet architecture and the key model is a small fully connected network. SurfEmb experiments with different embedding dimensions, mainly 6 and 12. The two models are trained together with contrastive loss, which will be explained later. With this loss, the key model is taught to map keys that are from similar, i.e. highly symmetric, object points close to each other in embedded space. In the cube example, this would mean the eight corners' keys are close in the embedded space. A pixel over a corner transformed by the query model will then land somewhere close to these keys, indicating it is strongly correlated to all of them.

6.3 The query and key models and their training

The establishing of dense correspondence distributions is the most novel part of SurfEmb. The authors believe they are the first to achieve this. Therefore the query and key model should be explained. The two models and their relationship can be seen in figure 6.1

6.3.1 The query model

The query model is the most complicated of the two. As mentioned, it has a ResUNet architecture which means it outputs one or multiple images of the original size. By first encoding the image, the model can detect object features and gain a latent understanding of the objects position and rotation. The decoding is then responsible for upscaling so each pixel in the original image can be correlated to an object point. The output is an image where each pixel is a vector in the embedded space.

6.3.2 The key model

The key model is simply a fully connected network trained on a specific object. It takes as input a 3D object point, and outputs a key in embedded space. It should be mentioned that the activation function used is the sine function. This is inspired by a 2020 paper[21] using it to great success for parameterizing images. That is, training a fully connected model to map pixel coordinates of a specific image to their RGB values. They showed that the sine function can give a much better mapping from pixel coordinates to pixel values than other popular activation functions. This is similar to what the key model is supposed to do. In our case we want to map the 3D coordinates of object points to their embedded values, essentially parameterizing our object.

6.3.3 The loss function

The two models are trained jointly using contrastive loss. To see why constrastive loss is used lets consider a simpler example. Figure 6.2 shows an example of a CNN embedding MNIST images to a vector space. The CNN is supposed to map images of the same digit close to each other in embedded space, and images of different digits should be far away from each other. This is similar to how the key and query models are supposed to embed object points and pixels close if they correspond. In the case of the MNIST dataset, each image is labelled with its class, i.e. digit. In that case, the network can simply be trained by with categorical cross entropy loss. However, in our case there are no labels for where in embedded space each pixel and object point should land. This is precisely what we want the model to learn on its own. This is where contrastive loss comes in.

Contrastive loss[8] was introduced in 2005 as a way to reduce the dimensionality of data while keeping as much information as possible. The loss can

be used to learn a new space of a desired dimension where the distance between datapoints is preserved as much as possible. SurfEmb experiments with a dimension of 6 for their embedded space. The information in the image must in our case be reduced to a dimension of 6. While a pixel in an RGB image has a dimension of 5, 2 of which for position and 3 for the RGB values, its dimension will increase when using information from other pixels, like when applying filters in a CNN. Therefore the query model should be seen as a mapping from a pixel from a high dimensional feature space to 6D, rather than a mapping from 5D to 6D. The key model on the other hand is a mapping from 3D object points to 6D keys, as it is a fully connected network that does not rely on filters or any information about other object points for its mapping.

While contrastive loss requires no labels to compare the output to, it does still require some prior knowledge of similarity, or distance, in the input space. The similarity to be preserved in our case is the correspondence between a pixel and an object point. Using the ground truth poses for each training image, this information is accessible. A pixel is considered similar to an object point if the pixel displays that point when the object is under the ground truth pose. A pixel and object point is considered dissimilar if the object point is sampled randomly from the object. This means the

Specifically, SurfEmb uses InfoNCE[17] loss, where NCE stands for Noise Contrastive Estimation. Let an embedded pixel be denoted as the query q and an embedded object point be denoted as the key k. Let $U = \{u_1, u_2, ..., u_N\}$ be a set of pixels sampled uniformly from the object mask in a training image and $\tilde{S} = \{c_1, c_2, ..., c_N\}$ be a set of uniformly sampled object points from the 3D model. The InfoNCE loss then looks like this:

$$L_e = -\frac{1}{|U|} \sum_{u \in U} \log \frac{\exp(q_u^T k_u)}{\sum_{c_i \in \tilde{S} \cup c_u} \exp(q_u^T k_i)} \tag{6.2}$$

Where k_u is the key of the object point c_u present at the pixel u. At the core of this loss function is the dot product between queries and keys. This is used as a measure of how well a query correlates to a key. A big dot product means high correlation. While the dot product is dependent on the angle between the vectors, it is also dependent on their lengths. Therefore the similiarity is not simply measured by the angle between the two vectors. The consequence of this will be explained shortly.

Also at the core of this loss function is the softmax function that generally looks like this:

$$\begin{aligned} y_i &= \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}} \\ x &= [x_0, x_1, ..., x_N] \end{aligned} \tag{6.3}$$

Where x is the input vector and y is the output vector. The softmax function is a popular way of normalizing a vector such that all elements in y are between

0 and 1, and the sum of all elements is 1. This property makes it popular for converting the output vectors of classification networks to probabilities. The softmax function also amplifies the relative differences between the elements. The larger the elements, the more their relative difference is amplified, as shown by the following example:

$$\begin{aligned} \frac{5}{5+4} &= 0.555 \\ \frac{e^5}{e^5+e^4} &= 0.731 \\ \frac{e^{10}}{e^{10}+e^8} &= 0.881 \\ \frac{e^{100}}{e^{100}+e^{80}} &= 0.999 \end{aligned} \tag{6.4}$$

The argument taken in by the softmax function is the dot product $q^T k$. The numerator takes in the dot product of the similar pair $q_u^T k_u$, and the denominator iterates through the dissimilar pairs and the similar pair. The function then gives a normalized measure of how close the similar pair are in relation to how close the dissimilar pairs are. Since softmax exaggerates the relative difference more as numbers increase, as shown in equation 6.4, a query with infinite norm will uniquely identify the key with the most similar angle. As the queries norm decreases, the softmax function no longer favors their dot product disproportionately above dot products with keys of similar angle, and so the pixel's distribution starts to spread over more object points. Essentially, the norm of the query determines how sure the model is at distinguishing between keys that are close to each other.

The result of the softmax can be seen as the probability that pixel u corresponds with object point c_u. This is a normal way of representing probabilities in classification models[22]. If this probability is already much higher than the others, then the loss function should not punish the model that much. If it is close to or lower than some of the dissimilar pairs then the model should punish it much more. Taking the logarithm achieves this property of the loss function, making training faster([22] pg.180). The negative sign is added to make the loss function strictly positive.

[17] showed that minimizing this loss function will result in estimating the following relation:

$$exp(q^T k_i) \propto \frac{p(c_i|q)}{p(c_i)} \tag{6.5}$$

Where the right side is the probability density function. Since c_i is sampled randomly and uniformly across the object surface, $p(c_i)$ will be constant. Therefore:

$$exp(q^T k_i) \propto p(c_i|q) = p(c_i|I, u, u \in M) \tag{6.6}$$

Since they are proportional, it is seen that an estimate of the probability function

in equation 6.1 can be found by normalizing $exp(q^T k_i)$ over the object surface:

$$p(c_i|I, u, u \in M) = \frac{\exp(q_u^T k_i)}{\iint_{c_j \in S} \exp(q_u^T k_j)} \tag{6.7}$$

Where S is the surface of the object. It shows that by minimizing the loss function we are training the query and key models to estimate the what we want, a distribution over the object surface for any given pixel u. The integral in reality is approximated by a summation over all other pairs (q_u, k_i) for $i \in \tilde{S}$. Where $\tilde{S}$ is a discrete set of sampled object surface points.

To summarize: for any pair q_u and k_i their correlation is measured by $exp(q^T k_i)$. To find the probability that q_u in fact corresponds to k_i, one must know how much it correlates with all other object points as well. This is why $exp(q^T k_i)$ must be normalized over all other object points.

Until now we have assumed that all pixels are a part of the object mask. The query and key models were trained on this assumption, as can be seen from the loss function in 6.2, which only uses pixels sampled from the object mask U. Consequently, the probability distribution in equation 6.7 also assumes that the pixel is a part of the object mask. This is of course not the case for all pixels. To compensate for this, SurfEmb adds another channel to their query model responsible for estimating the object mask. The channel will produce a discrete probability distribution giving the probability that each pixel contains the object:

$$Pr(u \in M|I, u) \tag{6.8}$$

This is then simply multiplied with the distribution in equation 6.7, creating a new distribution over correspondences:

$$p(u, c|I) = Pr(u \in M|I, u)p(c_i|I, u, u \in M) \tag{6.9}$$

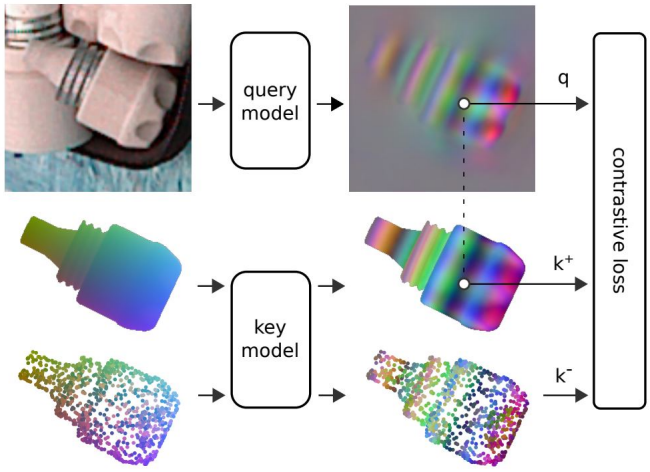
This new channel also needs a loss to learn the object masks, so an average binary cross-entropy loss is added to the total loss. This type of loss is a relatively simple loss used for classification models with only two categories([22] pg.187).

6.4 From distributions to poses

An overview of the process is shown in figure 6.3.

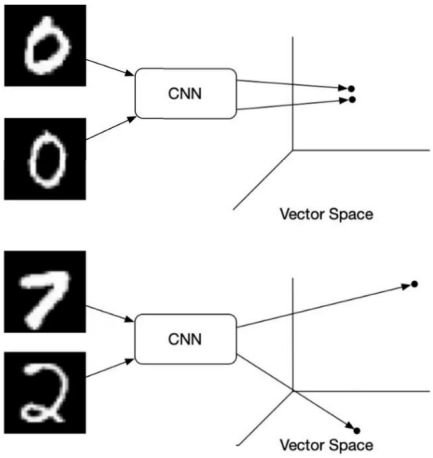
Each pixel now has a distribution over the object points. The distribution is stored in a 3D matrix where the first two axes are the pixel coordinates and the third axis contains the object points. This correspondence matrix is given by:

$$\boldsymbol{M_{corr}} = \begin{bmatrix} p(c|u_{00}) & \dots & p(c|u_{w0}) \\ \vdots & \ddots & \vdots \\ p(c|u_{0h}) & \dots & p(c|u_{wh}) \end{bmatrix} \tag{6.10}$$



Source: Taken from the SurfEmb paper[9]

Figure 6.1: How the two models are trained using contrastive loss. The negative and plus sign on the keys k indicate positive or negative keys. A positive key means similar to q and a negative means dissimilar. As shown the negative keys consist of a discrete set of random surface samples, whereas the positive key is just one point; the one displayed by the pixel in question.



Source: https://towardsdatascience.com/contrastive-loss-explaned-159f2d4a87ec

Figure 6.2: A CNN transforming MNIST images to a new vectorspace for comparison.

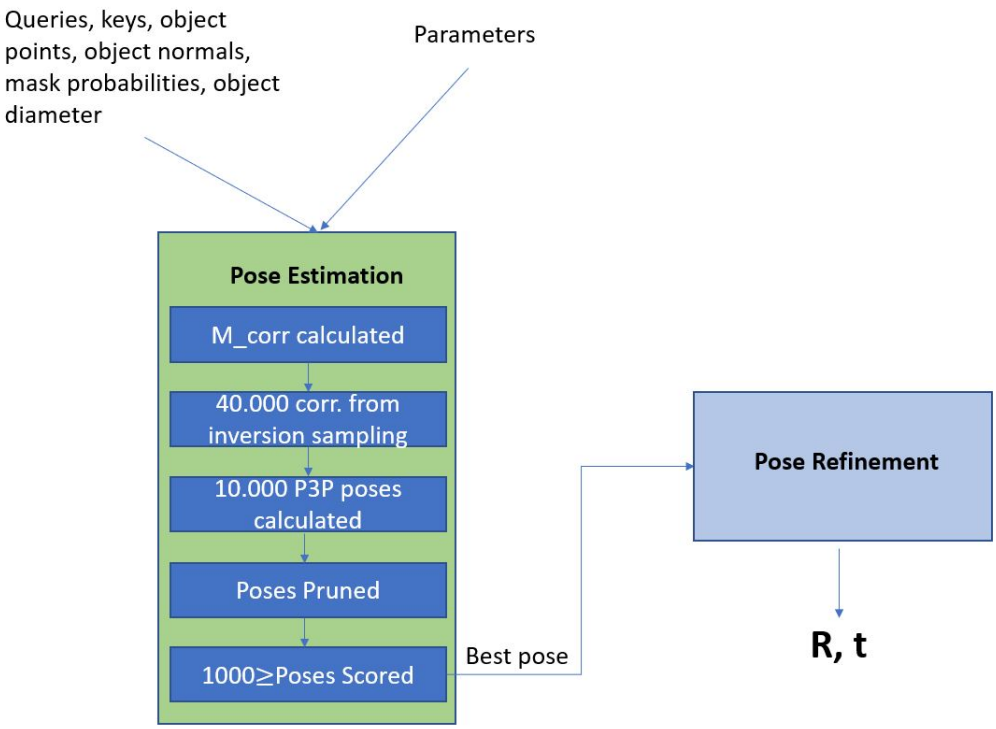


Figure 6.3: The pipeline from keys and queries to a final pose.

Where w and h are the width and height of the image given in pixels. $p(c|u_{xy})$ is a 1D vector containing the distribution over all object points for the pixel with coordinates (x, y). An element with index $[x, y, i]$ in this matrix gives the probability for a pixel with coordinates (x, y) to correspond to an object point c_i.

The PnP-RANSAC scheme uses the AP3P solver as its solver. Therefore four random correspondences must be selected in each iteration. For computational and memory efficiency, SurfEmb picks a set number of correspondences first before doing RANSAC. By doing this they can free the memory used by the correspondence matrix. They call this parameter *max_poses* and they set it to 10.000. This means they pick 40.000 correspondences before starting the RANSAC scheme.

In order to maximize the chance that each set of four points give a good pose estimate they should each be picked from areas of the matrix with high probability mass. SurfEmb increases the chance that such correspondences are selected by altering the representation and using inversion sampling. They do the following to achieve this:

First they unfold the matrix to a 1D vector:

$$\boldsymbol{v_{corr}} = \begin{bmatrix} p(c|u_{00}) & p(c|u_{10}) & \dots & p(c|u_{w0}) & p(c|u_{01}) & \dots & p(c|u_{w1}) & \dots & p(c|u_{wh}) \end{bmatrix} \tag{6.11}$$

They then make the distributions more extreme by raising the vector to a factor $\gamma = 1.5$. This reduces the entropy of the distributions as lower numbers are decreased relatively more than higher numbers. γ is a parameter that can be experimented with. The vector is then made cumulative, meaning each element is the sum of itself and all previous elements in the original vector. Finally the vector is divided by its last element. It is now a vector of monotonically increasing elements from 0 to 1.

Inversion sampling can then be done by selecting a random number between 0 and 1 and finding the first element in the vector with a greater value. Since the areas with high probability mass will cause the cumulative value to increase fast, a random number is more likely to land in these areas. An element in this vector gives the indices for a pixel and an object point which are then extracted from their respective arrays and saved to a new array for later use in P3P. They also save the face normal vectors for each object point to be used for later pose pruning.

The RANSAC procedure is then started.

6.4.1 Finding poses with RANSAC

SurfEmb uses a modified RANSAC scheme. Because their correspondences are distributions it is difficult to define a consensus set. They therefore skip steps 3, 4 and 5 in the list in section 3.4. They use a simpler approach where a number of minimal solutions are calculated and then scored based on the surface distributions for all pixels. Meaning they go directly from the minimum solution to the loss calculation on the whole dataset. The best scoring pose is then selected and further refined with local optimization of all distributions.

As is standard in PnP-RANSAC, the SurfEmb method uses the P3P solver as a sub-routine in their RANSAC scheme. This does however mean that there are up to four different solutions. One could randomly select one of the solutions every iteration, but this will lead to many iterations wasted on bad solutions. Therefore, SurfEmb chooses to use AP3P function as implemented in OpenCV[4]. This function takes in 4 points instead of 3, and uses the first 3 to find poses, and then the last point is used to extract the pose with the best reprojection error. Reprojection error in this case is the distance between the input image points and the projection of the corresponding object points under the pose hypothesis. If the P3P problem gives four unique solutions, the final point will probably have big reprojection errors for some of the solutions, which can then be rejected. If the fourth point does not agree with any of the solutions, then it becomes random which of the solutions has the best reprojection error, and so is still no worse than selecting one of the solutions at random.

All 10.000 sets of 4 are iterated through giving 10.000 poses. These poses must somehow be scored based on how well they fit with the probability dis-

tributions. The total score is based on a mask score 6.9 and the correspondece probability distributions. Scoring poses is an expensive calculation so the 10.000 poses are pruned first, removing poses that are very poor. These consist of poses where all correspondences come from a small area in the image, poses that are very close to or far away to the camera and poses where a correspondence is not visible as estimated by the face normals.

A parameter max_pose_score determines the maximum amount of poses to be scored, and is set to 1.000. Any post-prune poses exceeding this are thrown away. This is why an early termination RANSAC scheme can improve run-time without decreasing accuracy. This will be discussed some more later.

After the poses have been scored, the pose with the best score is selected and is refined by local maximization of the probabilities of the visible object points. This refinement process is also expensive, taking up to 30% of the time total run-time according to my results. The refined pose is the final output of SurfEmb.

Chapter 7

Experimentation

7.1 The Plan for experimentation

The original plan was to experiment with using a different method than PnP-RANSAC as the pipeline from probability ditributions to poses. The authors of SurfEmb state that the RANSAC scheme and pose scoring are the two bottlenecks for processing time. This was also found in my own experiments. RANSAC is an old method and is outperformed by newer methods such as Graduated Non-Convexity(GNC) when they are applicable. The advantage of RANSAC is that it is a very simple method that can be used or modified to fit many modelling problems. Since SurfEmb introduced novel surface distributions, the application of other methods than RANSAC has likely not been explored fully. The authors do not state why they chose RANSAC, and do not mention the possibility of using other methods in their "Improvements" section. I therefore had to ideate from scratch.

7.2 RANSAC substitution

I spent some time coming up with ideas on how RANSAC could be substituted, but with a limited timeframe I was not able to experiment with implementing any ideas. The problem also seemed challenging. A related paper that similarly creates many-to-many correspondences by dividing each object into a number of fragments[12] also choose to use a modified and improved version of PnP-RANSAC. This indicates that applying other outlier detection methods may be difficult for such correspondences.

The original task of my project was to compare GNC to Deep Learning for registration problems. Therefore I read a paper[23] about GNC and wondered if this can be a possible replacement for the RANSAC scheme. Since GNC is mainly used in situations where correspondences are one-to-one, not multimodal distributions, a reformulation or modification would be necessary. Additionally, there seems to be limited use of GNC in PnP problems as I found no papers

on this. I saw two possibilities, either the GNC algorithm had to be modified, or the probability distributions had to be sampled to create a one-to-one set of correspondences on which GNC or another robust non-minimal solver can be applied.

The former case may not be feasible because of the multimodal distributions. Most outlier rejection methods are based on one-to-one correspondences as these are most common. This includes GNC, which operates by first considering a square cost function with respect to the distance between each correspondence pair. The cost function is gradually made more non-convex, iteratively going closer to a truncated cost function. The gradients of correspondence pairs with long distance contribute less and less to the cost functions gradient for each iteration, essentially being ignored as outliers. When correspondences are distributions and are multimodal instead of unimodal, there is no single distance to measure. The loss function is instead the probability for a pixel-object point pair.

The second possibility is to create a set of one-to-one correspondences by sampling the distributions. Ideally one should choose correspondences in a way that respects symmetries, and not just picks random points as is the case with inversion sampling. Using the cube example, a pixel displaying a corner could for example become eight one-to-one correspondences, one for each corner. This would create many outliers, but many robust methods can deal with up to 90% outlier fractions[23], so this would need to be tested. For objects that are symmetric about and axis, like a bottle, this would not work as there are infinite possible correspondences, but for objects with few symmetries it becomes more realistic.

Ideally, the dimensionality of the distributions, which consists of a $wh*75000$ matrix, where w and h are the width and height of the input image, could be reduced to a format where picking out the best one-to-one pairs is easier. A simpler, but more encoded, representation of this matrix is already available as the embedded queries and keys. These have a dimension of $(wh + 75.000)E$, where E is the dimension of the embedding space. As discussed in section 6.4, the norm of a query can be a measurement how confident the model is at a precise location on the object. Therefore it may be possible to determine which correspondences are best suited to be a part of the one-to-one set by looking at the query norms. SurfEmb uses this when refining the depth of the pose when a depth image is available. They choose the set of pixels whose queries are at least 80% of the maximal query norm because they state it indicates the model is certain, and that it means those pixels likely correspond to visible parts of the object.

I spent some days trying to come up with ways to transform the distributions to a simpler format, but had to abandon the idea as it seemed difficult and would take a lot of time to implement ideas in code.

7.3 Improving RANSAC

My attention turned to improving the RANSAC scheme instead. The authors of SurfEmb stated that an early termination of the RANSAC method is a possible improvement. As discussed their RANSAC scheme first finds 10.000 poses before pose pruning to ensure that also difficult instances have enough post-prune poses for scoring. Through testing I found that the amount of poses pruned varies by image and object, but is usually around 40-95%. They then only score up to 1.000 poses as pose scoring is expensive, and any excess poses are discarded. If a low portion of the 10.000 poses are discarded by pruning, then it is unnecessary to find that many poses. Instead, one can calculate poses and prune them in batches and then continuously check if the number of post-prune poses is greater than 1.000. This possible improvement was mentioned by the authors in their paper. By doing this there cannot be any impact on accuracy, as the number of scored poses is equal in both cases. I implemented such a batch system and tested it. The code can be found in appendix A. Additionally their RANSAC scheme is dependent on some hyperparameters that I also experimented with.

The RANSAC parameters that were experimented with were:

- γ - The exponent to which the probabilities are raised to sharpen them. Defaulted to 1.5
- max_poses - Determines the number of correspondences picked and P3P poses calculated. If early termination is used, then this only decides the number of correspondences sampled and gives an upper bound for P3P poses calculated. Defaulted to 10.000
- pnp_batch_size - A parameter introduced by me. The number of P3P iterations done between each check of the number of post-prune poses.
- Object points - The amount of sampled points on the object surface. Defaulted to 75.000
- max_pose_evaluations - the maximum number of poses to be scored by the expensive scoring algorithm. Defaulted to 1.000.

7.4 Implementation

The SurfEmb source code was downloaded from their github(https://surfemb.github.io/) and studied. From the BOP website the *lmo* dataset was downloaded. *lmo* stands for Linemod-Occluded and is a dataset consisting of some textureless household objects with discriminatory colors and various levels of occlusion. Pre-trained key and query models trained on this dataset were available for download by the SurfEmb authors. They also provide a set of detection results from CosyPose for a set of *lmo* training images, which can be used for inference without needing to implement CosyPose. The source code, along with the dataset and pre-trained model were set up on NTNU's computer cluster IDUN.

After the setup was complete a model could be trained on a training set, and tested on the provided lmo inference set.

Due to limited experience with IDUN, deep learning in general, a limited timeframe and the fact that many models would need to be trained in order to find interesting results, I chose to download the pre-trained *lmo* model provided on the SurfEmb github. There were still experiments that could be done on the RANSAC part of the method.

7.5 Testing procedure

The dataset for inference provided on the github did contain some weird detections. Some of the bounding boxes were very bad and seemingly did not contain the object at all. This resulted in both the original SurfEmb and my test version failing and getting wildly wrong pose results. Luckily the detection dataset also contained a score for each bounding box which made it possible to skip all bounding boxes with bad scores. The scores were on a scale from 0 to 1. Initially, all bounding boxes with a score less than 0.8 were discarded as this seemed to be a safe value for getting accurate poses. The comparison between my test version and the original SurfEmb version were then conducted on 100-500 instances where the bounding box had a good score. The tests were always conducted on the same 100 or 500 instances, where an instance is essentially a task description containing the id of the target object, image id and the downloaded bounding box from the maskRCNN detection. All 8 objects in the *lmo* dataset were evenly included in the instances.

The SurfEmb source code only logs the runtime of the RANSAC method and the pose scores based on the ditributions. It does not contain a comparison to the ground truth of the inference data. In order to compare the performance of the different parameter configuration, a script had to be made that compares the results to the ground truth. I downloaded the ground truth data from the BOP website and implemented a script that loads the ground truth for each instance. A pose hypothesis and the ground truth consist of a rotation matrix and a translation matrix. In my experiments I only compared the rotation matrices. In order to compare the two, the Frobenius norm was used. This is a simple measure of the similarity between two matrices:

$$||A - B||_F = \sqrt{\sum_{i=0}^{n}\sum_{j=0}^{m}(a_{ij} - b_{ij})^2} \tag{7.1}$$

Where A and B are two matrices containing elements a_{ij} and b_{ij} respectively. The Frobenius norm is calculated between the ground truth rotation matrix and the results from the original and test version matrices respectively. The Frobenius norms are summed up for all instances and the total Frobenius norm is then used to compare the results.

Chapter 8

Results

The following results are from inference on a set of 100-500 *lmo* images, using pre-downloaded bounding boxes obtained by CosyPose's maskRCNN model.

The most interesting results are those from the early RANSAC termination as they give a runtime decrease at no cost to accuracy. For the following results, T_{est} is the pose estimation time, T_{ref} is the pose refinement time and F_{tot} is the sum of all Frobenius norms. The results of the early termination RANSAC scheme in isolation is show in table 8.1.

PnP Batch Size	50	100	200	400	1000	5000	10.000(Original)
T_{est}(s)	18.71	18.02	17.85	17.93	18.57	25.23	36.90
T_{ref}(s)	14.82	14.61	14.11	14.35	14.64	14.45	15.15
F_{tot}	43.02	45.40	40.65	39.85	43.69	41.22	42.31

Table 8.1: Comparison of different batch sizes over 100 instances. No other parameters were changes. Total estimation time, refinement time and the total Frobenius norm are listed.

The variation in Frobenius norms is entirely random due to the randomness of RANSAC. The estimation time sees approximately a 50% runtime decrease with a batch size less than 1000. A probable reason for why small batch sizes do not perform better is because of the overhead of pruning small batches rather than larger batches. Pruning is done on the GPU so it is more efficient when batches are large. There may also simply too much variation to detect the runtime decrease. Since the key/query models only take milliseconds to output their queries and keys, the total runtime of SurfEmb comes almost exclusively from the estimation and refinement. This means the early termination improvement gives approximately a 38% decrease in overall runtime. This should generalize beyond the 100 instances on which it was tested.

Reducing the amount of object points also gave a runtime decrease at little cost to accuracy. I found that by increasing gamme to 2.5 the number of total P3P poses calculated over 100 and 500 instances decreased, meaning fewer poses

were pruned.

I started experimenting with a minimum boundig box score of 0.8. Tables 8.2 and 8.3 show the results I got after trying to optimize the parameters in combination with early termination.

Version	Test	Original
T_{est}(s)	23.65	181.15
T_{ref}(s)	67.96	70.81
F_{tot}	221.56	219.97

Table 8.2: Over 500 instances with minimum bounding box score 0.8. Only altered parameters are shown. Original parameters are given in parenthesis. Parameters: PnP Batch Size = 50(10.000), max pose evaluations=500(1.000), keys = 20.000(75.000), gamma=2.5(1.5)

Version	Test	Original
T_{est}(s)	18.88	184.20
T_{ref}(s)	70.08	71.53
F_{tot}	221.19	219.07

Table 8.3: Over 500 instances with minimum bounding box score 0.8. Only altered parameters are shown. Original parameters are given in parenthesis. Parameters: PnP Batch Size = 50(10.000), max pose evaluations=500(1.000), keys = 10.000(75.000), gamma=2.5(1.5)

I later decreased the minimum acceptable bounding box score to see if the accuracy of my test version decreased. This means that some new instances previously skipped are now included. Table 8.4 shows results with a minimum bounding box score of 0.6.

Version	Test	Original
T_{est}(s)	19.72	180.39
T_{ref}(s)	69.00	69.92
F_{tot}	241.72	245.87

Table 8.4: Over 500 instances with minimum bounding box score 0.6. Only altered parameters are shown. Original parameters are given in parenthesis. Parameters: PnP Batch Size = 50(10.000), max pose evaluations=500(1.000), keys = 10.000(75.000), gamma=2.5(1.5)

Clearly the total Frobenius norm has increased, indicating poorer pose estimates by both the original version and my test version. The test version still gives accurate results even though it scores less poses and uses less object points.

Finally, to see if my test version can compete when bounding boxes are very good I increased the bounding box score to 0.98 and 0.999. The results are shown in tables 8.5 and 8.6.

Version	Test	Original
T_{est}(s)	3.61	37.26
T_{ref}(s)	14.03	14.16
F_{tot}	24.84	23.23

Table 8.5: Over 100 instances with minimum bounding box score 0.98. Only altered parameters are shown. Original parameters are given in parenthesis. Parameters: PnP Batch Size = 50(10.000), max pose evaluations=500(1.000), keys = 10.000(75.000), gamma=2.5(1.5)

Version	Test	Original
T_{est}(s)	2.74	30.50
T_{ref}(s)	10.18	10.30
F_{tot}	26.15	26.47

Table 8.6: Over 100 instances with minimum bounding box score 0.999. Only altered parameters are shown. Original parameters are given in parenthesis. Parameters: PnP Batch Size = 50(10.000), max pose evaluations=500(1.000), keys = 10.000(75.000), gamma=2.5(1.5)

Showing that the alterations I have tested can uphold a competitive accuracy with both good, and not so good, bounding boxes.

With just 20.000 keys and no change to the downsampling, this also uses less memory.

Increasing max pose evaluations gave little to no increase in accuracy.

Chapter 9

Discussion

The results show that the runtime decreases significantly with minimal or no impact to accuracy.

The early termination RANSAC improvement yielded significant run-time improvement on the tested dataset. The run-time decrease is dependent on the relative difficulty disparity of the images in a dataset. This is because for datasets with only easy images one could instead lower the max_poses parameter down from 10.000. For datasets with ambiguities such as cluttering and occlusion, some instances can be very difficult while others are easy. The 10.000 poses may then be needed for the difficult instances where more poses are pruned, but adds inefficiency to the easy ones. This is where early termination can improve run-time the most. Therefore I believe the run-time decrease should generalize well for more difficult datasets. Additionally, the tests were conducted on synthetic images. In real world images there may be more noise. Therefore in real world applications the max_poses parameter should be a high number to ensure robustness. The early termination improvement can allow for an increase in max_poses without a cost to run-time.

The results of parameter tuning is more limitied. The testing was limited to the *lmo* dataset. SurfEmb was submitted by the original authors to be tested on other datasets as well, so they likely did not tune the parameters to fit the *lmo* dataset specifically. These are parameters that should be adjusted based on the dataset. Therefore the tuning is not exactly an improvement, but rather experimentation to test the limits of SurfEmb on one specific dataset. If these results generalize to other datasets has not been tested. The results are also limited to a specific set of images since all testing was done on the same set. However, by looking at 500 images and varying the minimum bounding box score I believe it to be likely the results generalize well for the entire *lmo* dataset.

Another limitation is the measurement of accuracy. The total Frobenius norm was used, which may not be the optimal way to measure accuracy. It may also not be the optimal way to compare rotation matrices.

Chapter 10

Conclusion

The theory needed to get a basic understanding of SurfEmb has been covered. The novelty of SurfEmbs work and how it was achieved has been explained. Some ideation of improvements by substituting the RANSAC scheme, and the challenges thereof, have been briefly discussed. SurfEmb was experimented with by the altering of some parameters and one definite run-time improvement was introduced through early termination of the RANSAC loop.

Bibliography

[1] Herbert Bay et al. "Speeded-Up Robust Features (SURF)". In: *Computer Vision and Image Understanding* 110.3 (2008). Similarity Matching in Computer Vision and Multimedia, pp. 346–359. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2007.09.014. URL: https://www.sciencedirect.com/science/article/pii/S1077314207001555.

[2] *BOP: Benchmark for 6D Object Pose Estimation*. URL: https://bop.felk.cvut.cz/datasets/. (accessed: 18.12.22).

[3] François Chollet. *Deep Learning with Python*. Manning, Nov. 2017. ISBN: 9781617294433.

[4] *cv.SolvePnP*. URL: https://amroamroamro.github.io/mexopencv/matlab/cv.solvePnP.html. (accessed: 18.12.22).

[5] Lowe DG. "Object recognition from local scaleinvariant features". In: *In: IEEE International Conference on Computer Vision (ICCV)* (1999), pp. 1150–1157.

[6] Foivos I. Diakogiannis et al. "ResUNet-a: a deep learning framework for semantic segmentation of remotely sensed data". In: *CoRR* abs/1904.00592 (2019). arXiv: 1904.00592. URL: http://arxiv.org/abs/1904.00592.

[7] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: https://doi.org/10.1145/358669.358692.

[8] R. Hadsell, S. Chopra, and Y. LeCun. "Dimensionality Reduction by Learning an Invariant Mapping". In: 2 (2006), pp. 1735–1742. DOI: 10.1109/CVPR.2006.100.

[9] Rasmus Laurvig Haugaard and Anders Glent Buch. "SurfEmb: Dense and Continuous Correspondence Distributions for Object Pose Estimation with Learnt Surface Embeddings". In: *CoRR* abs/2111.13489 (2021). arXiv: 2111.13489. URL: https://arxiv.org/abs/2111.13489.

[10] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[11] Stefan Hinterstoisser et al. “Gradient Response Maps for Real-Time Detection of Textureless Objects”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.5 (2012), pp. 876–888. DOI: 10.1109/TPAMI.2011.206.

[12] Tomas Hodan, Daniel Barath, and Jiri Matas. “EPOS: Estimating 6D Pose of Objects with Symmetries”. In: *CoRR* abs/2004.00605 (2020). arXiv: 2004.00605. URL: https://arxiv.org/abs/2004.00605.

[13] Tong Ke and Stergios I. Roumeliotis. “An Efficient Algebraic Solution to the Perspective-Three-Point Problem”. In: *CoRR* abs/1701.08237 (2017). arXiv: 1701.08237. URL: http://arxiv.org/abs/1701.08237.

[14] Yann Labbé et al. “CosyPose: Consistent multi-view multi-object 6D pose estimation”. In: *CoRR* abs/2008.08465 (2020). arXiv: 2008.08465. URL: https://arxiv.org/abs/2008.08465.

[15] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. “EPnP: An accurate O(n) solution to the PnP problem”. In: *International Journal of Computer Vision* 81 (Feb. 2009). DOI: 10.1007/s11263-008-0152-6.

[16] Yi Li et al. “DeepIM: Deep Iterative Matching for 6D Pose Estimation”. In: *CoRR* abs/1804.00175 (2018). arXiv: 1804.00175. URL: http://arxiv.org/abs/1804.00175.

[17] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. “Representation Learning with Contrastive Predictive Coding”. In: *CoRR* abs/1807.03748 (2018). arXiv: 1807.03748. URL: http://arxiv.org/abs/1807.03748.

[18] 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: *CoRR* abs/1703.10896 (2017). arXiv: 1703.10896. URL: http://arxiv.org/abs/1703.10896.

[19] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[20] F. Rothganger et al. “3D object modeling and recognition using local affine-invariant image descriptors and multi-view spatial constraints”. In: *International Journal of Computer Vision (IJCV)* (2006), pp. 231–259.

[21] Vincent Sitzmann et al. “Implicit Neural Representations with Periodic Activation Functions”. In: *CoRR* abs/2006.09661 (2020). arXiv: 2006.09661. URL: https://arxiv.org/abs/2006.09661.

[22] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch*. 2020. URL: https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf.

[23] Heng Yang et al. “Graduated Non-Convexity for Robust Spatial Perception: From Non-Minimal Solvers to Global Outlier Rejection”. In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 1127–1134. DOI: 10.1109/LRA.2020.2965893.

[24] Yinqiang Zheng et al. “Revisiting the PnP Problem: A Fast, General and Optimal Solution”. In: (2013), pp. 2344–2351. DOI: 10.1109/ICCV.2013.291.

Appendix A

The early RANSAC termination Python code:

In [ ]:

```
#This code is only a slight modified version of the original SurfEmb source code.
#The original can be found at https://github.com/rasmushaugaard/surfemb/blob/master/surfemb/pose_est.py.
#Contribution: Batching the P3P pose process.
# wrapping a while loop around the P3P solver and pruning to continually check total number of post-prune poses.
# Terminate when >min_post_prune_poses

_t = 0  # counts total post prune poses
k = 0 # number of batches iterated
pn = pnp_batch_size # Number of P3P poses calculated per batch

#counters used for testing:
n_p3p_no_solution = 0
n_distz_prune = 0
n_norm_prune = 0
n_distimg_prune = 0

batch_n = max_poses//pn
with timer('pnp+prune', debug):
    while _t<min_post_prune_poses and k<batch_n:
        poses_batch = np.zeros((pn,3,4))
        p2d_batch, p3d_batch, n3d_batch = np.zeros((pn,3,2)), np.zeros((pn,3,3)), np.zeros((pn,3,3))
        poses_mask = np.zeros(pn, dtype=bool)

        #print("running while loop it:", k)
        assert(max_poses%pn==0)
        p3d_batch = p3d[k*pn:(k+1)*pn] # Batch of 3D object points
        p2d_batch = p2d[k*pn:(k+1)*pn] # Batch of 2D image points
        n3d_batch = n3d[k*pn:(k+1)*pn] # Batch of object surface normals
        with timer('pnp', debug):
            rotvecs = np.zeros((pn, 3))
            for i in range(pn):
                ret, rvecs, tvecs = cv2.solveP3P(p3d_batch[i], p2d_batch[i], K, None, flags=pnp_method) #Finding poses
                if rvecs:
                    j = np.random.randint(len(rvecs))
                    rotvecs[i] = rvecs[j][:, 0]
                    poses_batch[i, :3, 3:] = tvecs[j]
                    poses_mask[i] = True
                else:
                    n_p3p_no_solution += 1
            poses_batch[:, :3, :3] = Rotation.from_rotvec(rotvecs).as_matrix()
        poses_batch, p2d_batch, p3d_batch, n3d_batch = [a[poses_mask] for a in (poses_batch, p2d_batch, p3d_batch, n3d_batch)]

        with timer('pose pruning', debug):
            # Prune hypotheses where all correspondences come from the same small area in the image
            dist_2d = np.linalg.norm(p2d_batch[:, :3, None] - p2d_batch[:, None, :3], axis=-1).max(axis=(1, 2))  # (max_poses,)
            dist_2d_mask = dist_2d >= dist_2d_min * res_sampled
            n_distimg_prune += len(poses_batch) - dist_2d_mask.sum()

            # Prune hypotheses that are very close to or very far from the camera compared to the crop
            z = poses_batch[:, 2, 3]
            z_min = K[0, 0] * obj_diameter / (res_sampled * 20)
            z_max = K[0, 0] * obj_diameter / (res_sampled * 0.5)
            size_mask = (z_min < z) & (z < z_max)
            n_distz_prune += len(poses_batch) - size_mask.sum()

            # Prune hypotheses where correspondences are not visible, estimated by the face normal.
            Rt = poses_batch[:, :3, :3].transpose(0, 2, 1)  # (max_poses, 3, 3)
            n3d_cam = n3d_batch @ Rt  # (max_poses, 3 pts, 3 nxnynz)
            p3d_cam = p3d_batch[:, :3] @ Rt + poses_batch[:, None, :3, 3]  # (max_poses, 3 pts, 3 xyz)
            normals_dot = (n3d_cam * p3d_cam).sum(axis=-1)  # (max_poses, 3 pts)
            normals_mask = np.all(normals_dot < 0, axis=-1)  # (max_poses,)
            n_norm_prune += len(poses_batch) - normals_mask.sum()

            # allow not pruning for debugging reasons
            if do_prune:
                poses_batch = poses_batch[dist_2d_mask & size_mask & normals_mask]  # (n_poses, 3, 4)
                poses = np.concatenate((poses,poses_batch),axis=0)
                _t = len(poses)
        k+=1
```