Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Adrian Gjertsen

Regressing 6D Pose Using a Symmetry-Aware Intermediate Representation

A Combination of SurfEmb and GDR-Net

Master's thesis in Produktutvikling og produksjon
Supervisor: Olav Egeland
June 2023

NTNU
Norwegian University of
Science and Technology

Adrian Gjertsen

Regressing 6D Pose Using a Symmetry-Aware Intermediate Representation

A Combination of SurfEmb and GDR-Net

Master's thesis in Produktutvikling og produksjon
Supervisor: Olav Egeland
June 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Contents

List of Figures

List of Tables

Abstract

In this masters thesis the possibility of speeding up a pose estimation method called SurfEmb[8] by using deep learning is investigated, inspired by another method, the current state of the art GDRNPP. GDRNPP is an improvement of an earlier method called GDR-Net[21], and of the two only GDR-Net has a paper available. SurfEmb uses deep learning to find a novel correspondence representation by embedding pixels and object points to a shared embedded space, enabling them to represent symmetries of the object. From the embeddings they build correspondences as probability distributions and use a PnP-RANSAC[5] scheme to find a pose, the latter of which is very time consuming. GDR-Net regresses a more traditional one-to-one correspondence representation, and solves the issue of symmetries by also regressing a separate symmetry map. The symmetry map is a less compact way of representing symmetries than SurfEmbs embeddings. It uses these two as input in a pose regression model to find a pose, which is orders of magnitude faster than SurfEmbs PnP-RANSAC scheme. In this thesis, a combination of the two methods is tested by regressing a pose from SurfEmbs embeddings, which to the best of my knowledge has not been done before. The results indicate that my method is a 20x speedup over SurfEmb, but the average rotation and translation error are between 2-4x greater than SurfEmb and GDR-Net. The training process and model architecture in this thesis likely has multiple short-comings that when fixed will increase the accuracy. Internal testing on a self-made model indicated that regressing a pose from SurfEmbs representation performs equally to regressing from traditional dense correspondences like used in GDR-Net. Whether this holds true for better tuned and trained models is not clear.

Chapter 1

Introduction

The 6 Degree of Freedom(6D) pose estimation problem is the problem of finding the rotation and translation of an object in an image. The object of interest is stored as a 3D object on the computer, and has a defined local coordinate system. The objective is to find the transformation matrix between the cameras coordinate system and the objects local coordinate system as configured in the image.

There are different versions of this problem depending on what data is available. Some methods are aimed at having multiple views of the object, taken by a moving camera or different cameras. These are called multiple view methods. With multiple views of the same scene the depth ambiguity of image formation can be eliminated by for example triangulation. Depth can also be directly measured using a depth sensor in addition to a RGB camera. Having depth information will make the translation estimation easier. Different methods focus on different versions of the problem. This thesis will focus on the single-view RGB version, where there is only one RGB image available with no depth information. This is the most difficult version since it contains the least information about the scene.

Methods also differ in their use of deep learning. All current methods use deep learning to some extent, but the extent of which varies. There are currently two main categories. Those in the first category use a two stage approach. They use deep learning to find correspondences between points on the object surface and pixels in the image. By knowing enough such correspondences it is possible to mathematically calculate a pose estimate. The problem of finding a pose from 2D-3D correspondences is called a Perspective-n-Point(PnP) problem, and can be solved in multiple ways, some examples are [13, 14, 24]. Earlier methods, before deep learning, would use algorithms such as SURF[1] to find a few easily detectable keypoints in the image. These are called sparse correspondences. With deep learning it is more common to regress dense correspondences, meaning each pixel in the image is correlated to a point on the object. An image where each pixel has values equal to its corresponding object point coordinate is called a dense correspondence map.

Methods in the second category are called end-to-end methods since they use deep learning from beginning to end. They usually have two networks, one to transform the input image into a new and simpler image, and a second network to regress a pose from the simplified image. The simplified image could be a dense correspondence map, which is what GDR-Net uses, or some other representation an engineer feels is helpful for the pose regression model. Some of the end-to-end methods are called correspondence-free methods since they do not create any correspondences. Only in the last couple years have end-to-end methods achieved similar accuracy to the ones solving PnP problems.

SurfEmb[8] is a 6D pose estimation method based on using single-view RGB images. Using deep learning, they find a novel representation of correspondences, a probability distribution over the objects surface. Meaning, for a given pixel, each point on the objects surface has a probability of corresponding to that pixel. This means a pixel can correspond to multiple parts of the object, meaning the model can display uncertainty in the face of symmetry. In order to find a pose they sample the probability functions in a way where highly probable correspondences are most likely to be picked, and use PnP-RANSAC to find a pose. SurfEmb was in 2021 the highest scoring method on the BOP website, which is a hub meant to facilitate competition between 6D pose estimation methods.

Since 2021 SurfEmb has been overtaken by 5 other methods on the BOP website. The current leader is an end-to-end method called GDRNPP, uploaded to the BOP website in october 2022. GDRNPP is a improved version of GDR-Net[21] with some optimizations shortly stated on the BOP website, but uses the same architecture. There exists only a paper for GDR-Net, published 24. February 2021.

This thesis will take a look at how SurfEmb and GDR-Net function and investigate whether the two methods can be combined to create a better method. Initially, in order to understand these methods, the relevant theory of image formation, deep learning and Convolutional Neural Networks(CNNs) will be covered. The relevant parts of SurfEmb and GDRNPP will then be explained in detail, followed by the motivation for combining them. The method by which they were combined and tested will then be explained. Finally the experimental results will be presented and discussed.

Chapter 2

Motivation

The goal of this work was to use SurfEmb as a basis for an improved pose estimation method. For my project thesis in the fall semester of 2022 I sped SurfEmb up by 20-30% by slightly modifying their RANSAC[5] procedure. I also experimented with hyper-parameters in their RANSAC procedure to try and optimize them. Modifying the parameters achieved speed-ups with little or no detriment to accuracy, but did not add anything novel to the method. The hope for this masters thesis is to achieve a much more significant speed-up by replacing the RANSAC procedure with a deep learning model. A deep learning model will utilize the GPU much more efficiently and can regress a pose in milliseconds, whereas the RANSAC loop runs thousands of iterations on the CPU, taking somewhere in the area of 10-100x longer. The hope was to get a significant speed increase with little or no detriment to accuracy. The idea of using a deep learning model for finding a pose was inspired by the current state of the art method GDRNPP. Essentially the idea was to combine the beginning half of SurfEmb with the second half of GDRNPP. How and why will be discussed in detail in a later chapter.

Chapter 3

Theory

In order to understand the various methods discussed in this thesis it is important to have a basic understanding of deep learning, projective geometry and some camera theory. As stated in the introduction, some methods rely solely on machine learning in order to find a pose. Others use a two stage approach, first using machine learning to gain an understanding of the image and then formulate a Perspective-n-Point(PnP) problem based on 2D-3D correspondences. How PnP problems can be solved will not be covered as it is not used in this thesis. The project thesis I wrote last semester covers this and is found in the appendix. Projective geometry and camera theory will still be relevant for how the deep learning model estimates depth, and so is covered.

3.1 Projection and Camera Theory

Section 3.1 is a revised version of the camera theory section in my project thesis, which is found in the appendix.

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.

3.1.1 The Pinhole Model

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, without being distorted. This hole is called the center of projection(COP). This is shown in figure 3.1. Of course, no real cameras have an infinitely small aperture. Real cameras use lenses to increase the light received.

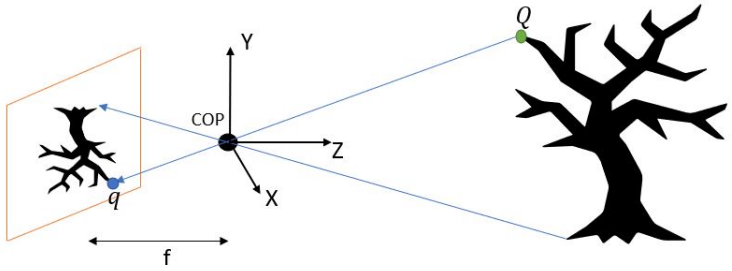


Figure 3.1: The pinhole model. The camera coordinate system is drawn. The image plane is in reality behind the COP, as shown, and the projected shape is upside down. It is often flipped and drawn in front of the COP to make the math more elegant. The focal length is marked f.

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 pose estimation they can be neglected.

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. A 3D point in the world becomes a 4D homogeneous coordinate, and a 2D image coordinate becomes a 3D homogeneous image coordinate.

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

$$q = \boldsymbol{P}Q \tag{3.1}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \boldsymbol{P} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.2}$$

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.3}$$

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 3.1. If the distance to the image plane along the Z axis is defined to be 1, then by dividing q by a constant such that its third element is equal to 1, the point of intersection is found.

$$q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X\frac{1}{Z} \\ Y\frac{1}{Z} \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3.4}$$

Note that any 3D world coordinate can also be seen as a 2D homogeneous image coordinate. Simply divide it by its z-component and its projection onto an image plane with focal length 1 is found. This is very simple but not very useful for practical purposes when using cameras. It is more useful to know the pixel coordinates of the projected point, so it can be located in the digital image.

In reality, the image plane has a specific distance from the COP given in real world 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.

$$\boldsymbol{K} = \begin{bmatrix} f\alpha_u & 0 & u_0 \\ 0 & -f\alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

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 from homogeneous 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:

$$\boldsymbol{K}q = \boldsymbol{K} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Xf\alpha_u + Zu_0 \\ -Yf\alpha_v + Zv_0 \\ Z \end{bmatrix} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{3.6}$$

$[u, v]$ 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 relevant for how a neural network can understand cropped and zoomed images.

3.2 Convolutional Neural Networks

This section assumes some basic knowledge of how machine learning functions. Specifically how a neural network consists of layers of neurons transforming inputs with weights, biases and activation functions, and how the parameters are updated in the backwards pass. To limit the scope of this thesis these concepts will not be explained here. The project thesis I wrote last semester covers these basic things and can be found in the appendix. Some aspects, like convolutions and overfitting, that are especially relevant to this thesis will be covered. Grander concepts like network architectures will also be covered.

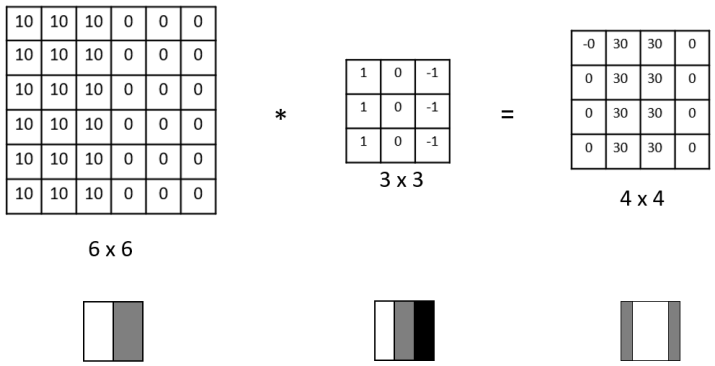
3.2.1 Images as Input

The first version of neural networks was the Multi Layer Perceptron(MLP). It consists of multiple layers of neurons where each neuron is connected to every neuron in adjacent layers. Meaning two layers of 4 will have 16 connections. The number of parameters will increase quadratically with the size of the layers. This works well when the input and output size is of a small dimension. For images however this quickly becomes very unefficient. An RGB image has an input dimension equal to the number of pixels times three, meaning the number of connections grows very large. This means the model grows large in size and might use too much memory to fit on a device.

Not only is it memory inefficient but the architecture does not lend itself to understanding images. Images contain local information that may not always be in the same location. To understand objects in an image, a neural network must have what is called translation invariance. That is, it must understand that for example an airplane is an airplane no matter where in the image it is. An MLP will learn that airplanes exist only where they exist in the training data. The airplane may also be closer to the camera, which will confuse the network since its size has changed. To fix the issue of model size and translation and size variation, Convolutional Neural Networks(CNN) were introduced.

3.2.2 Convolutions

A convolution in deep learning is a transformation of an image using a filter. A filter is a quadratic matrix of weights which is sequentially shifted over the image, multiplying each overlapping value and placing the sum of the products in a new matrix. The process is shown in figure 3.2. The network will update the weights of the filters as it learns. As shown in the figure, individual filters are able to detect low level features like corners and edges. The output of a convolution is therefore called a feature map. By shifting sequentially over the

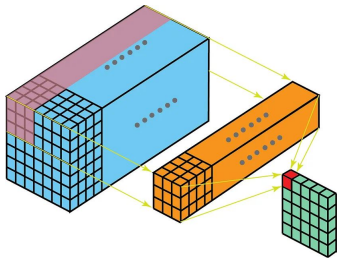


Source: https://medium.com/analytics-vidhya/convolutional-neural-networks-cnn-a78e78c1ba94

Figure 3.2: A 3x3 filter is convolved with a 6x6 input image. The 3x3 filter starts in the top left of the original image, overlapping with the top left 3x3 sub-matrix of the 6x6 image. The overlapping values are multiplied together element wise, similar to a dot product of two vectors. This gives nine products, which are then summed together. The result is placed in the top left of the 4x4 output image. The filter then shifts until each 3x3 sub-matrix has been covered once. The filter in this figure is able to detect edges, meaning the output image has high values where an edge is detected in the original image.

whole image the network can detect shapes anywhere in the image, becoming translation invariant.

The filter and images shown in figure 3.2 are two dimensional. Usually an image is three-dimensional, using the extra dimension to store colors. An RGB image is essentially three images, each storing the levels of red, green and blue respectively. The images along this dimension are called channels, meaning an RGB image has three channels. In order to convolve with a three dimensional image, a filter must also have three dimensions. A filter of spatial size 3x3 would in this case be a 3x3x3 filter, meaning there are 9 overlapping values as the filter shifts over the image. The sum of all nine products are put in the output image, meaning the output still only has two dimensions. This is illustrated for an arbitrary number of channels in figure 3.3. Usually multiple filters are used on an image, and so the number of channels in the output will be equal to the number of filters used.



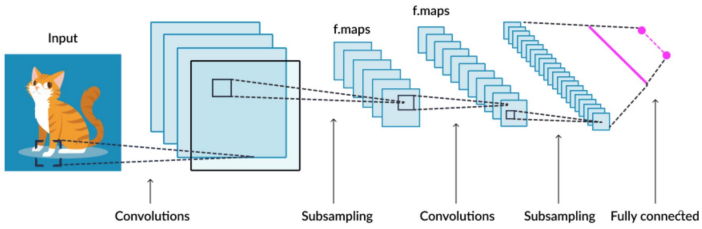
Source: https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37

Figure 3.3: A 2D convolution done over multiple channels.

By chaining convolutions, a network can gain higher level feature detections. With enough layers it can learn to detect entire objects like cats, bikes, pedestrians etc. Next, the architecture of a typical CNN will be discussed.

3.2.3 CNN Architecture

Figure 3.4 shows a typical architecture of a CNN built to detect objects. Multiple convolutions are chained together, each followed by a sub-sampling process. Finally the output is small enough to be processed by a fully connected layer, also called an MLP.



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

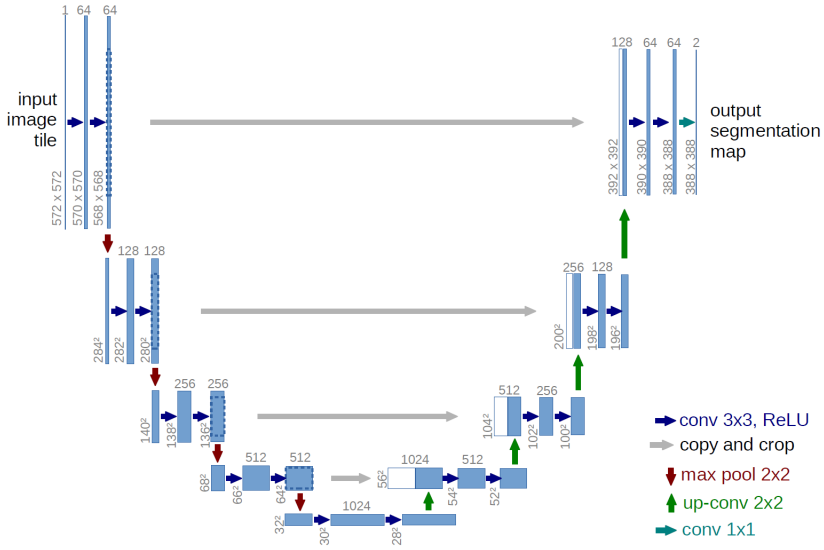
Figure 3.4: A typical CNN structure. The feature maps are down-sampled between each convolution to reduce spatial size. In the end the network has some fully connected layers to reduce the output dimension.

As shown in the figure, the CNN has a sub-sampling process after each convolution. This is employed to reduce the spatial dimension of the images so that subsequent filters can detect larger features and so that the output eventually can be fed into an MLP. The task of the MLP is to condense the 3D feature map to, usually, a 1D vector. For detection purposes the final dimension might be only a single number between 0 and 1 indicating if a certain object is visible in the image.

A widely used sub-sampling method is max-pooling. Similar to convolutions, a window is slid over an image, but instead of taking the dot product, the largest value in each channel is selected and stored in the output. The number of channels stays the same for the input and output, but the spatial size is decreased based on the size of the filter. Usually a 2x2 filter is used, which will decrease both spatial dimensions by a factor of 2. Figure 3.4 indicates that the number of channels increases during sub-sampling, but this does not happen during pooling. The number of channels can change if a strided convolution is instead used for sub-sampling, but this wont be used or covered in this thesis.

3.2.4 Encoder-Decoder Architecture

In some situations it is more useful to have the model output be an image instead of a number. For example if the model is meant to highlight something in the image. U-Net[18] is a CNN architecture that was developed to segment an image. The U-Net architecture is shown in figure 3.5. Segmenting means finding segments of the image that fit into some predefined class, and highlighting them in the original image. It is used in autonomous driving, where the locations of objects is very important. Figure 3.6 shows an example of a segmented image.



Source: Taken from [18]

Figure 3.5: A typical U-Net architecture. The network uses convolutions to encode the image, gradually decreasing its spatial size and increasing the number of feature maps in order to understand the contents of the image. The decoder is responsible for up-scaling the information so that the position of features can be displayed in an output image. The grey arrows are feed-forward connections meant to help the network up-scale by giving it a view of previous layers.

Source: https://medium.com/visionwizard/object-segmentation-4fc67077a678

Figure 3.6: A segmentation of a scene used for autonomous driving. The network has transformed the original image into a new image which the cars logic system can use to determine an action.

U-Net has many variations so there is a general term for such architectures: encoder-decoder network. SurfEmb uses a variation called ResUNet[4], which uses residual connections as presented in [9] in a U-Net architecture. Residual connections are skip-connections that append the input of a layer to its output. These connections provide a path for the gradient to flow through without going through activation functions which can cause the gradients to explode or vanish. Exploding and vanishing gradients will be discussed in the next section.

Most methods within pose estimation use a encoder-decoder architecture. For the ones relying on finding correspondences the encoder-decoder is used to regress an image where each pixel contains a 3D coordinate. Such images are called dense correspondence maps, as each pixel is paired to an object point. For end-to-end methods it is used to find some simplification of the original image, from which it is easier to regress a pose. The pose regression model will be similar to the one in figure 3.4, condensing an image to a rotation and translation prediction.

3.3 Aspects to Consider During Training

How a deep learning model is trained is very important for its performance. There are many aspects to consider when training a network, some major ones will be covered here.

3.3.1 Hyperparameters and Optimizers

There are a number of parameters that specify some aspects of the training loop.

- Model size - The size of the model, measured in the number of parameters,

determines the complexity it can achieve in the input-output transformation.

- Batch size - determines the size of each batch during training. This will influence how stochastic the calculated gradient is. A small batch is likely to be less representative of the entire training set than a larger batch. Since the gradients of each item in the batch are summed up before the network steps, a small batch size means the gradient is more stochastic. This can actually be a very good trait in optimization in general, as it means the network is less likely to get stuck on local minima. Having too much randomness can make it hard for the model to converge. Therefore batch size must be experimented with. A batch size between 10 and 30 is usual.

- Learning rate - a constant determining the step size of the parameter update. The learning rate is a parameter used when creating the optimizer object. A small learning rate generally yields slower, but more stable learning.

The optimizer object takes as input the training loss and is responsible for updating the parameters of the model. Perhaps the most basic optimizer is the Stochastic Gradient Descent(SGD) optimizer, which simply multiplies the learning rate with the gradient to determine step size. Other optimizers use more complicated calculations to find the best step size and direction, such as Adam[11]. The Adam optimizer takes into consideration the moving average of the gradients when calculating the step direction.

3.3.2 Overfitting

One of the important things to avoid when training a model is overfitting. Overfitting is when the model specializes too much on the training images, losing generality and performance on validation and test images. The model begins to rely on memorizing the training data, including the noise that should ideally be ignored. Similar to how a student might memorize a physics formula as a sequence of letters instead of understanding the physical and logical meaning behind the formula. This can happen if the model is too complex, that is, it has more parameters than the task requires, or if the model is trained for too long on too few training images.

A simple example of this is the following: Consider a model trained to predict the temperature in Celsius given the temperature in Kelvin. The relation is obviously linear, but the model does not know this. If the engineer designing the model does not have the insight to see that the relationship is simple, they may make a complex model capable of representing high order polynomials. Given $n - 1$ 2D points, an n-degree polynomial can be found intersecting all data-points in the training data. Since the training data will contain some inaccuracies, there will never be a linear line intersecting all the data-points. Therefore, in order to minimize the loss function, the model may find the high

order polynomial instead of the linear line. The polynomial will obviously not give good results outside of the datapoints in the training set. The model is in that case said to be poor at generalizing and to be overfitting.

There are multiple ways of combatting overfitting. Three of which will be discussed next.

3.3.3 Data Augmentation, Model Complexity and Dropout

The amount and quality of training data is very important for the performance of a deep learning model. With a low amount of training data it is harder for a model to get a general understanding of the inputs. This is especially the case if the inputs are complex and of high dimension, like images. With higher amounts of training data it is harder for the model to memorize the training data, meaning it is forced to generalize. It is also very important that the training data does not have a bias that is not present in the target data the model is supposed to be used on. The model may learn to rely on the bias and be confused by its absence when employed in the real world. The relation between the amount of training data, its quality and models complexity will determine if the model overfits or not. A complex model is analogous to a high order polynomial as used in the above example. For complex problems a complex model is necessary, so in order to decrease overfitting it may be necessary to increase the amount of training data or use methods like data augmentation and dropout.

The optimal way of increasing training data is to somehow get more real, high quality data. In our case that would mean taking more photos of the objects. In many situations this is not feasible. Therefore it is normal to simulate a larger training data by augmenting the existing data. Augmentation can be any alteration of an existing datapoint. For example for a detection model it is normal to rotate the training images randomly, or mirror them. Noise can be added to the images to vary how each neuron is activated, making the model more robust and less likely to overfit. Augmentations can remove some of the bias in the training data by introducing randomness. For example in this thesis the models will be trained on synthetically rendered images, which will have different lighting, textures and colors than a real image. By randomly augmenting the colors and introducing some noise, the network hopefully will be better prepared for real images. The augmented data should still represent the target data the model is intended to work on, and so how the data should be augmented can vary from use case to use case.

Another popular way of combating overfitting is to use regularization techniques such as dropout or weight decay. These measures attempt to combat overfitting by changing how the network functions during training. A dropout layer will randomly deactivate a percentage of neurons in the following layer. With dropout, the information flows differently each time, meaning it is less likely the network falls into the trap of specializing too much on unimportant details in the training data that are not represented in test data. Dropout is done only during training. Weight decay is a term added in the loss function

that penalizes the network for having large weights and biases. The idea is that having a few large weights and biases that dominate the other parameters gives poor generalization.

If the model still tends to overfit then the model might be too complex. A less complex model should be tested. For a CNN this would mean either reducing the number of convolutional layers, or reducing the number of channels produced by each convolution. The network is then either made shorter or thinner.

3.3.4 Slow and Unstable Training

Other central problems when training a model are unstable and slow training. Unstable training could mean the training loss suddenly starts increasing and spirals out of control. Slow training means the model takes too long to decrease the loss. One of the first things to check in those cases is the learning rate of the model. Increasing the learning rate can increase the learning speed, but can make it more difficult for the network to find the minima in parameter space due to overshoot. In the worst case it will make the model unstable. Decreasing the learning rate will make the training more stable, but could slow training down drastically. If the learning rate is reasonable and there is still slow or unstable learning, it is more helpful to directly address the cause of the slow or unstable learning. Often the problem is vanishing or exploding gradients, or poorly scaled inputs. Luckily there exists solutions that address each of these issues.

Vanishing Gradient and its Solutions

A vanishing gradient means the gradients becomes close to zero when calculated in the backwards pass, meaning the network barely updates its parameters each step, causing slow learning. This can occur if too many of certain activation functions are chained. For example the sigmoid function will saturate for large or small inputs as shown in figure 3.7, causing very small derivatives. Multiplying many such small derivatives together in the backwards pass will cause vanishingly small gradients. Because of this the ReLU function is popular in deep learning. It has a derivative equal to 1 for all positive inputs, meaning the gradient is retained along paths with positive values. For negative inputs the derivative is zero, which means for some datapoints many of the ReLU neurons will be outputting 0 and not updating. This is normal behaviour, but can be problematic if the networks weights update in such a way that a ReLU neuron *always* sees negative inputs, for all training data. Then the neuron is irreversibly dead since it can never be updated. Despite this one flaw, ReLU is considered to be the most popular activation function in deep learning and is used in for example GDRNPP.

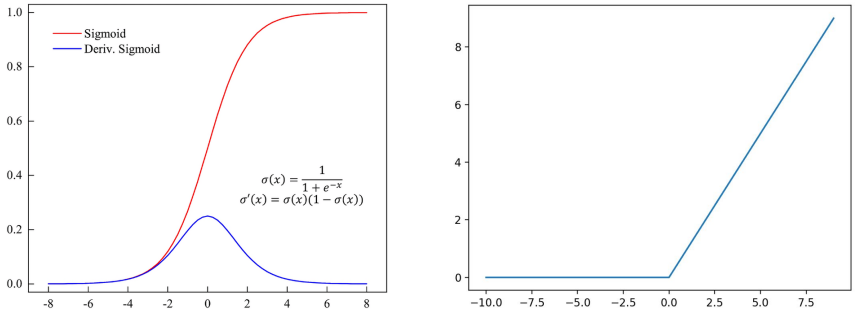
Exploding Gradients

The backwards pass can also result in very large or very small(very negative) gradients. This happens if many neurons along the gradient path have large weights. The weights will be multiplied together which causes exponential growth with respect to the number of layers. ReLU activation functions in particular can cause this problem as it is unbounded in the positive domain. A common sign of exploding gradients is an unstable training loss, or a training loss that grows to infinity. How much a parameter is changed in each iteration is proportional to its gradient, and so while the model will try to reduce the training loss by updating the weights, it does so with too large steps, likely worsening the problem. As previously mentioned when describing ResUNet, residual connections are also used to combat the issue of vanishing or exploding gradients.

Poorly Scaled Inputs

For efficient learning it is important that the training data has a good format and scale. It is often regularized to having zero mean and unit variance. This is done by calculating the mean and variance of the training data, and then rescaling it. Say for example a network is to be trained to predict the price of a car given its age and kilometers driven. The age will be on the scale of ones and tens, whereas the kilometers driven can be many thousands. Because of the difference in scale, the network will need to be very sensitive to the number of years compared to the number of kilometers. The network might overestimate the importance of kilometers driven because of the larger scale. This can also make learning slow as the network takes time to increase or decrease its weights to rescale the inputs. By doing this beforehand, the network does not need to learn how to rescale the data, which saves a lot of time and makes finding the loss minima easier.

It turns out it is also helpful to regularize the input between each layer as well. In deep networks in particular, the scale of the outputs in the deeper layers can vary between each batch, which means the network is chasing a moving target. By regularizing between each layer the network is stabilized. This also helps to combat the exploding gradient problem, as large values are scaled down. This method is called Batch Normalization[10] and is extremely popular in deep networks. Since the mean and variance of the entire training data is unavailable between each layer, Batch Normalization instead calculates the mean and variance of the current batch between each layer. Of course, during inference when the batch size is 1, regularization becomes impossible. Therefore, the Batch Normalization layers keep a running mean and variance from training which is used during inference.



(a) (b)

Figure 3.7: (a): The sigmoid function(red) and its derivative(blue). (b): The ReLU function.

3.4 Pose Estimation of Symmetric Objects

Symmetries can make pose estimation more difficult. Lets say for example the object of interest is a perfect, textureless cube. Given an image of the cube there are 32 possible rotations that could yield the same image. The neural networks used to predict poses only output one single pose, and so it must choose one of the 32. Getting just one of the 32 poses as output is not the problem. Each of the 32 predictions are valid and correct for any practical intent and purpose, as the cubes properties do not change in any way depending on which of the 32 poses it is configured in. The problem instead stems from the issues that arise during training. If a model guesses one of the 32 correct rotations, it should not be penalized. If the ground truth for the image specifies only one of the 32 rotations and the loss function is based on the difference between the predicted and ground truth pose, the network will be confused. It is being asked to output different rotations for the same input image. In the training data provided by BOP there exists just one ground truth pose per image. Therefore it is necessary to implement a symmetry aware rotation loss, which GDRNPP does. They simply change the ground truth pose to be the one closest to the predicted pose.

There is also a problem with regressing dense correspondences. If the model is asked to identify each corner in the image, there will be eight alternatives. The model must choose just one. This can for example cause the model to map multiple corners in the image to the same corner on the object, which obviously violates the structure of the cube. Regressing a pose from a correspondence map with such errors can be difficult. GDR-Net solves this by regressing a separate symmetry aware map which will be covered in the GDR-Net chapter.

Symmetry is a grey area. There are objects that are not truly symmetric, but very close. For example an eggbox has a notch where the box is opened, meaning it is not symmetric. However it can be difficult for a CNN to identify the notch in certain situations, for example under occlusion. This gives large

discontinuity in the input-output transformation. The model is asked to predict a completely different rotation based on perhaps just a few pixels. Therefore it is up to the engineer to determine if the object can be assumed to be symmetric based on the use case.

3.5 The BOP Challenge

Within the deep learning field there are often public challenges or competitions for scientists to compete in. Usually these have public leaderboards and are meant to help advance the various fields within deep learning. For the pose estimation problem there exists a challenge called the BOP challenge which holds a yearly competition for the best pose estimation method. The website is found at: https://bop.felk.cvut.cz/.

The BOP website provides seven different datasets for which the methods can be scored on. These include the LM, LMO and TLESS datasets which are discussed in this thesis. The LMO dataset is a variant of the LM dataset with more occlusion, and will be used for training and testing in this thesis. The datasets contain somewhere between 5-30 different objects with various amounts of occlusion and textures. Real or synthetic training images for each dataset can be downloaded. They consist of RGB and depth images of a scene containing all or many of the objects randomly scattered on a table or on the ground. Often objects will occlude each other. If a method uses depth information it will be labeled RGBD and if it only uses the RGB images it will be labeled RGB. The organizers then score each method using a hidden test set, unavailable to the competitors.

In 2021 the most accurate method was SurfEmb. In 2022 it was surpassed by some other methods, most notably GDRNPP which is a big improvement on both runtime and accuracy. The decrease in runtime comes from the use of an end-to-end deep learning model instead of the two stage approach used by SurfEmb, where deep learning is used to find correspondences which are then used in a PnP-RANSAC scheme to find a pose. Interestingly there is also an increase in accuracy by using a deep learning model for pose regression rather than mathematically from correspondences.

The next chapters will cover the SurfEmb and GDRNPP methods and explore how the good parts from each of them can be combined for a possibly better method.

Chapter 4

SurfEmb

This chapter is a revised version of the SurfEmb chapter in my project thesis from the fall semester. It is included in this master thesis for the sake of completeness. The poject thesis can be found in the appendix.

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 website. 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, as of june 6th 2023. The RGB and RGBD versions are the same, the only difference is that the RGBD version refines the final pose translation using depth information.

4.1 The Basics

SurfEmb is based on finding 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[12] 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:

$$p(c|I, u, u \in M) \tag{4.1}$$

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

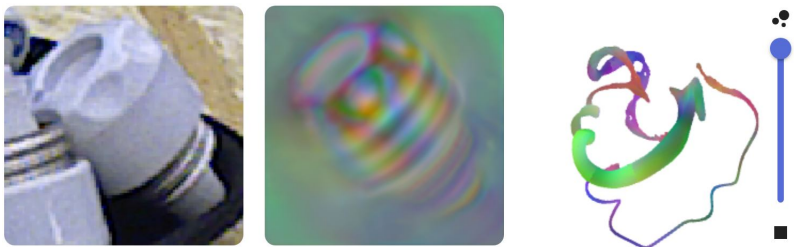
represent correspondence to more than just one object point. Taking a perfect, textureless cube as an example, this would in theory mean each pixel on a corner gets a 12.5% likelihood score on each of the eight corners. This is a multimodal distribution, and the authors claim that, to the best of their knowledge, SurfEmb is the first method to be able to represent such distributions. Figure 4.1 shows an example of this

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 some objects, 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 search for this pose, SurfEmb uses a P4P solver in a scheme similar to RANSAC. A P4P problem is a variation of PnP where only 4 correspondences are used. 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 P4P 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 in that pose. The surviving poses are sent through a scoring function that scores each pose based on their 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 function.

4.2 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 will embed the pixels and object points to a shared embedded space with 12 dimensions. 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. By training the models together using a version of constrastive loss called InfoNCE[17], the key model will learn to map symmetrical object points to the same place in embedded space. Since a large dot product equals a large correspondence, and dot products are largest when two vectors have the same orientation, the query model will learn to orient an embedding towards its corresponding key. If a query points towards a group of closely positioned keys, it indicates a correspondence with multiple object points, likely symmetrical or close on the 3D object. The length of the query embedding indicates how sure the model is at



Source: https://surfemb.github.io/

Figure 4.1: From left to right: The input RGB crop; an RGB visualization of the query image; and a 3D visualization of the object surface points embedding. By hovering over the images with the mouse pointer(white circle), the red coloration shows the probability distribution for a pixel. The image on the left shows how the symmetrical object results in a multi-modal distribution, indicated by multiple red dots. The query image shows that symmetries are colored with the same color. The object surface embedding shows how an object with an axis of symmetry collapses into a 1D string in embedded space.

distinguishing between closely grouped keys. Why this is will become clearer when looking at the formulas in the loss section below.

Approximately 75.000 object points are sampled from the object surface before training and inference. For ease of understanding it is recommended to take a look at the visualizations given on SurfEmbs website: https://surfemb.github.io/. They show how symmetries collapse onto the same points in embedded space. For example objects with a continuous symmetry along an axis, like a cylinder, will collapse to a one dimensional string in embedded space. An example of this is TLESS object number 2. Figure 4.1 shows an example from their website. The distribution for a given pixel can be calculated based on its query and the 75.000 keys. The equations for doing so will be explained later.

4.3 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 4.2

4.3.1 The Query Model

The query model is the most complicated of the two. It has a ResUNet architecture which means it outputs one or multiple images. By first encoding the

RGB crop, 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 transformed to a query. The output is a three-dimensional image where each pixel is a vector in the embedded space. SurfEmb chooses to use a shared encoder for all objects, but use a separate decoder for each object.

4.3.2 The Key Model

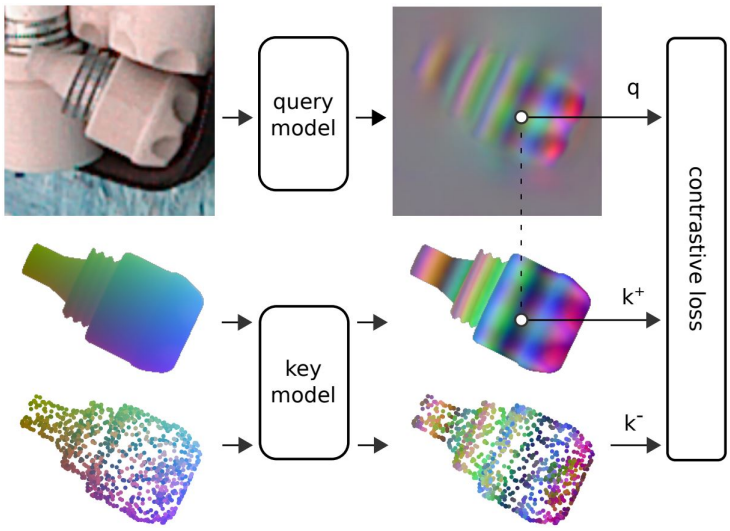
The key model is simply a fully connected network trained on a specific object. It takes as input a 3D object point, and outputs a key in embedded space. It should be mentioned that the activation function used is the sine function. This is inspired by a 2020 paper[19] using it to great success for parameterizing 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.

4.3.3 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 4.3 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 with categorical cross entropy loss. For categorical models the desired transformation is known beforehand as a vector where the correct class has maximal value. However, in our case, where in the embedded space a key and query should land is not known beforehand. This is precisely what we want the model to learn on its own. This is where contrastive loss comes in.

Contrastive loss[7] was introduced in 2005 as a way to reduce the dimension 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.

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 point of this loss was to preserve some similarity in a lower dimension. 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 in the ground truth pose. A



Source: Taken from the SurfEmb paper[8].

Figure 4.2: An overview of how the query and key models are trained. For a given training image, the key model takes in all the visible object surface coordinates in the mask, and also some randomly sampled points from the object surface. These are called positive and negative keys respectively. Contrastive loss should ensure that a query and its corresponding positive key result in a large dot product, while a query and a negative keys result in low dot products.

pixel and object point are considered dissimilar if the object point is sampled randomly from the object. That is, the object point is likely not displayed by the pixel. Figure 4.2 gives a visualization of this.

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:

$$L_e = -\frac{1}{|U|} \sum_{u \in U} \log \frac{\exp(q_u^T k_u)}{\sum_{c_i \in \tilde{S} \cup c_u} \exp(q_u^T k_i)} \tag{4.2}$$

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:

$$\begin{aligned} y_i &= \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}} \\ x &= [x_0, x_1, ..., x_N] \end{aligned} \tag{4.3}$$

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:

$$\begin{aligned} \frac{5}{5+4} &= 0.555 \\ \frac{e^5}{e^5+e^4} &= 0.731 \\ \frac{e^{10}}{e^{10}+e^8} &= 0.881 \\ \frac{e^{100}}{e^{100}+e^{80}} &= 0.999 \end{aligned} \tag{4.4}$$

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 4.4, a query with a very large 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[20]. If the probability for a similar pair is much higher than the probability for negative pairs, then the loss function should not penalize the model that much, if at all. If it is close to or lower than some of the dissimilar pairs then the loss function should penalize it much more. Taking the logarithm achieves this property of the loss function, making training

faster([20] pg.180). The negative sign is added to make the loss function strictly positive. To summarize: The loss function in equation 4.2 is a combination of: a correspondence measure using dot products, the softmax for converting to probabilities, and the logarithm function for better loss properties. The following is a mathematical proof that the infoNCE loss yields probability distributions.

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

$$exp(q^T k_i) \propto \frac{p(c_i|q)}{p(c_i)} \tag{4.5}$$

Where the right side is a probability density ratio. If an object point was rarely seen as a positive key during training, i.e. low $p(c_i)$, the model might hesitate to give it large probability, so dividing by $p(c_i)$ supposedly compensates for this. In this case though, since c_i is sampled randomly and uniformly across the object surface, $p(c_i)$ will be constant. Therefore:

$$exp(q^T k_i) \propto p(c_i|q) = p(c_i|I, u, u \in M) \tag{4.6}$$

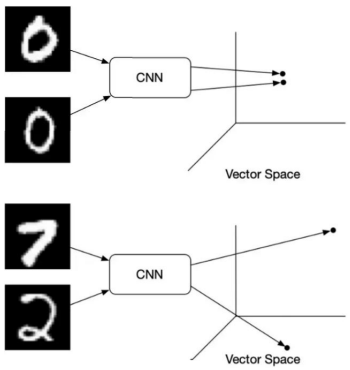
Since they are proportional, it is seen that an estimate of the probability function in equation 4.1 can be found by normalizing $exp(q^T k_i)$ over the object surface:

$$p(c_i|I, u, u \in M) = \frac{\exp(q_u^T k_i)}{\iint_{c_j \in S} \exp(q_u^T k_j)} \tag{4.7}$$

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 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, as shown in equation 4.2.

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 4.2, which only uses pixels sampled from the object mask. Consequently, the probability distribution in equation 4.7 also assumes that the pixel is a part of the object mask. This is of course not the case for all pixels. Under testing the model must be able to handle pixels outside of the object mask as well. Pixels outside the object mask should not correspond to any particular part of the object. Ideally the query produced from these pixels result in a high entropy distribution, giving essentially zero score to each object point. From my testing this seems to be the case automatically when using infoNCE loss, probably because the model learns to produce low norm queries when it finds no correlation in order to minimize loss. To ensure that such pixels have low probabilities, SurfEmb adds another channel to their query model responsible



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

Figure 4.3: A CNN transforming MNIST images to a new vectorspace for comparison.

for estimating the object mask. The channel will produce a discrete probability distribution giving the probability that each pixel contains the object:

$$Pr(u \in M|I, u) \tag{4.8}$$

This is then simply multiplied with the distribution in equation 4.7, creating a new distribution over correspondences:

$$p(u, c|I) = Pr(u \in M|I, u)p(c_i|I, u, u \in M) \tag{4.9}$$

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([20] pg.187).

4.4 From Distributions to Poses

For this thesis the specifics of the RANSAC procedure is not relevant. It is only important to note that it takes a lot of time and runs mostly on the CPU. In short they use inversion sampling to sample the probability functions for 40.000 correspondences. These are grouped to 10.000 groups of four and used to find 10.000 pose candidates using a P3P solver. The pose scoring highest using a score function based on the probability of each pixel-object point pair is selected for further refinement.

In the SurfEmb paper they state that in their experiments, inference time is approximately 2.2s. Of which 20ms are from regressing the query image, 1.2s from the RANSAC procedure and 1.0s from a pose refinement procedure. The RANSAC procedure is covered in detail in my project thesis found in the appendix.

Chapter 5

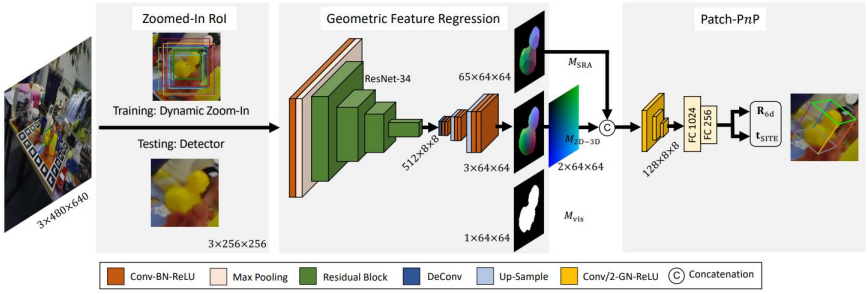
GDR-Net

When looking at the current top performing methods on the BOP database, GDRNPP is a clear winner. Unfortunately, the creators have not yet released a paper, but state that the method is mostly just an improvement of an earlier version from 2021, for which a paper does exist. This earlier method is called GDR-Net. GDR-Net stands for Geometry-Guided Direct Regression Network. GDR-Net is based on finding dense correspondences with an encoder-decoder architecture and then using a simple convolutional model to regress the pose. Since the models are consecutive the method is end-to-end trainable. The architecture is shown in figure 5.1.

5.1 Model Architecture

The architecture has some intermediate representations which the model is explicitly trained to produce. While it is possible to exclude these intermediate representations simply by removing their respective loss terms when training, they effectively serve as guides making the model learn more easily[21]. The idea is that these intermediate representations help the model focus on the object of interest, and make the job easier for the following Patch-PnP module. The intermediate maps shown in figure 5.1 consist of:

- M_{vis} - This is the mask of the visible parts of the object. A pixel is white if the object is visible at that pixel. This gives the network an understanding of which pixels belong to the object.
- M_{2D-3D} - This 5 channel image contains the dense correspondences. It is a combination of a 3D object coordinates image, called M_{3D}, and a 2D pixel coordinate image. The object coordinates are regressed by the encoder-decoder network and give the object coordinate displayed by a pixel, given in the objects local coordinate system. If a pixel is outside the object mask the coordinate is set to (0,0,0). Hence it is a 3xwxh image. The 2D pixel coordinate image is simply an image where each pixel has a



Source: Taken from [21]

Figure 5.1: The architecture of GDR-Net, taken from their paper. The original image is shown on the left. For training, a dynamic and randomized zoom-in is used to zoom in on the object of interest, as its position is known from the ground truth information. For testing the zoom-in must be done by a separate detector. The network first uses an encoder-decoder network to regress some intermediate images, and then uses the Patch-PnP network to regress a pose.

value equal to its position in the image, i.e. the top left pixel has values (0,0), the pixel to the right has values (1,0) etc. This map is always the same and does not need to be regressed. The 2D pixel coordinate image is often called a positional encoding map, as it encodes the position of each pixel. While the network functions without this 2D encoding, it has been shown to give slightly better performance when added in other methods[3], and a significantly better performance when added in GDR-Net.

- M_{SRA} - This is a 65 channel image and is a novel addition. This is meant to help the network understand symmetries. Before any learning is done the object is split into 64 fragments using furthest point sampling. Furthest point sampling samples 64 evenly distributed points on the object surface. These points serve as the centers of 64 fragments. Each fragment is represented by one channel in M_{SRA}. Each pixel in M_{SRA} has 64 dimensions and contains the probabilities of it belonging to each of the 64 fragments. Meaning if an object is a cube with eight corners, a pixel displaying a corner will ideally have a 1/8 score for each of the eight fragments containing a corner, and zero probability for all other fragments. M_{3D-2D} has no way of displaying such uncertainty as it can only assign each pixel to one 3D coordinate. M_{3D-2D} might therefore be confused by symmetric points and assign a single object point to multiple pixels, which can make learning harder for the subsequent pose regression network. Therefore M_{SRA} is added to give the network an understanding of such ambiguities before heading into the pose regression. They state "We utilize MSRA as a symmetry-aware attention to guide the learning of Patch-PnP". The authors also state that M_{SRA} eases the learning of M_{3D-2D} as it learns

to first regress coarse regions and then finer coordinates.

5.2 Parameterization of Rotation and Translation

5.2.1 Rotation

There are multiple ways of representing a rotation. Many of them are not unique, meaning there can be multiple ways of representing the same rotation. Taking Euler angles as an example, a rotation $\theta = \pi$ about the x axis yields the same pose as a rotation $\theta = -\pi, -3\pi, -5\pi...$ about the x axis. This is problematic for a regression network since there are multiple correct answers. Therefore most methods use unique representations where this is never the case.

Another problem is that representations with 4 or fewer dimensions have discontinuities in Euclidian space. To combat the first issue with multiple correct answers one might constrain the angles to $[0, 2\pi]$. This still has a problem, which is the discontinuity around zero rotation. A slightly positive rotation has a value close to 0, whereas a slightly negative rotation has a value close to 2π. This discontinuity makes it hard for the network to learn. Therefore it is necessary for a representation which solves both of these issues. For this purpose [25] introduces a 6 dimensional representation, which has achieved great success and is used by GDR-Net.

The representation works by letting the network regress a 6 dimensional vector from which a rotation matrix is constructed. The 6D vector contains the first two columns of the rotation matrix, not necessarily with unit length.

$$\boldsymbol{R_{6D}} = \left[\boldsymbol{r_1}|\boldsymbol{r_2}\right] \tag{5.1}$$

When given a vector $\boldsymbol{R_{6D}} = \left[\boldsymbol{r_1}|\boldsymbol{r_2}\right]$ the rotation matrix can be constructed as follows:

$$\boldsymbol{R_1} = \phi(\boldsymbol{r_1}) \tag{5.2}$$

$$\boldsymbol{R_3} = \phi(\boldsymbol{R_1} \times \boldsymbol{r_2}) \tag{5.3}$$

$$\boldsymbol{R_2} = \boldsymbol{R_1} \times \boldsymbol{R_3} \tag{5.4}$$

where ϕ represents vector normalization and $\boldsymbol{R_i}$ is column number i in the rotation matrix.

5.2.2 Translation

The GDR-Net paper states that directly regressing the translation does not work well in practice and therefore many methods choose to regress

the 2D location of the object center (o_x, o_y) and the distance t_z separately. Where o_x and o_y are given in pixel coordinates and t_z is given in real world units. The translation can then be calculated as:

$$\boldsymbol{t} = \boldsymbol{t_z}\boldsymbol{K^{-1}}[o_x, o_y, 1]^T \tag{5.5}$$

where $\boldsymbol{K}$ is the camera intrinsic matrix, which when inverted transforms the pixel coordinates into homogeneous image coordinates. These are then scaled by the real world value t_z to find the translation in world coordinates.

A separate detector is often used to find a bounding box of the object of interest, which is then zoomed in on and fed to the network. This can be seen in 5.1. There is a problem when feeding the network with zoomed-in images of the object. The network cannot regress the distance t_z without knowing how zoomed in the input image is.

To solve this issue, GDR-Net uses Scale Invariant Translation Estimation(SITE)[15] which is a parameterization meant to help the network understand zoomed in Regions of Interest(ROIs). It works by letting the network regress o_x and o_y normalized with respect to the image size, and regress t_z normalized to the zoom-in amount. Let the height and width of the bounding box be w and h, given in pixels, and $s_0 = \max(w, h)$. Let s_{zoom} be the dimension of the zoomed in ROI, which is always square, given in pixels. The ratio $r = s_{zoom}/s_0$ is then a measure of how zoomed in the ROI is. Let the center of the bounding box be $[c_x, c_y]$, which are known values. The network then regresses:

$$\boldsymbol{\delta_x} = \frac{o_x - c_x}{w} \tag{5.6}$$

$$\boldsymbol{\delta_y} = \frac{o_y - c_y}{h} \tag{5.7}$$

$$\boldsymbol{\delta_z} = \frac{t_z}{r} \tag{5.8}$$

The above three equations are used to find the predicted o_x, o_y and t_z. The predicted t is then found from equation 5.5 using the intrinsic matrix of the cropped and zoomed-in image rather than the original image.

5.3 Loss Functions

After constructing the predicted rotation matrix and translation vector, GDR-Net uses the following loss functions:

$$\boldsymbol{L_R} = \underset{x \in M}{\text{avg}} ||\tilde{\boldsymbol{R}}\boldsymbol{x} - \hat{\boldsymbol{R}}\boldsymbol{x}||_1 \tag{5.9}$$

$$\boldsymbol{L_{center}} = ||(\tilde{\delta}_x - \hat{\delta}_x, \tilde{\delta}_y - \hat{\delta}_y)||_1 \tag{5.10}$$

$$\boldsymbol{L_z} = ||\tilde{\delta}_z - \tilde{\delta}_z||_1 \tag{5.11}$$

Where the tilde and hat symbols represent predicted and ground truth values, x is a 3D surface coordinate and M is the 3D CAD model. The rotation loss is the average error of object points when rotated with the predicted rotation versus the ground truth rotation. The translation losses are simply the L_1 distances of the scale invariant parameters.

Chapter 6

Combination

The combination will be done by forming query images using SurfEmb, and then regressing the pose from the query image using a model similar to Patch-PnP from GDR-Net.

The motivation for combining these two models, and why I thought it to be a good idea, is that the SurfEmb query image may be able to serve the same purpose as M_{3D-2D} and M_{SRA} at the same time, while using less channels. As explained the M_{3D-2D} map is confused by symmetries, which is why GDR-Net also needs M_{SRA}. SurfEmbs embeddings are already aware of symmetries and will map symmetric points to the same values in its embedded space. Meaning there is never a situation where the subsequent pose regression network can be confused as it is always shown symmetries in the same consistent way.

The central question is if the pose regression network achieves the same accuracy when inputted with the query image as opposed to the dense correspondences. Is the query image as easy to understand as the dense correspondences of M_{3D-2D}, which for a human at least makes a lot more sense? On the other hand, they are essentially both just a re-coloration of the object. The 3D coordinate map gives a smooth coloration of the object, whereas the query image gives a 12 dimensional coloration with patterns, both of which shown in figure 7.1. It is difficult to say without experimentation whether the pose regression model prefers one over the other. To investigate this I tested four different methods varying the use of queries and coordinate maps. They will be explained in the next chapter.

Chapter 7

Method

Coding all the required code from scratch, without experience, would be far too much work. Since the proposed method essentially is an extension of SurfEmb, it is much more efficient to use their code. SurfEmb's data-loader, model class, image augmentations, renderer, embedding loss calculation and training loop are used with some modifications and additions. The SurfEmb source code was downloaded from GitHub (https://github.com/rasmushaugaard/surfemb). All code is written in Python and relies on the PyTorch library. Training and testing is doen on objects in the LMO dataset.

The IDUN computer cluster at NTNU is used for training and testing the models.

7.1 Pose Regression Model

The main purpose of this thesis was to use deep learning to speed up SurfEmbs pose estimation. The idea was to regress a pose directly from the query image. To test if this is possible I created my own pose regression model, taking inspiration from GDR-Net's PatchPnP. My model is found in the appendix. Using the PyTorch library in Python I created a custom model class. How to create custom models is explained in the book "Learning with PyTorch"[20]. With a custom model I can have full control over the forward function, which is needed to condense the output of the convolutional layers into rotation and translation predictions. It also enables me to use third party libraries like DropBlock[6] in the forward function.

I experimented with the architecture of the pose regression model. I chose to use five convolutional layers, with an input size of 256x256. The number of channels increases from 12 to 256 during the five convolutions.

Since Patch-PnP uses three convolutional layers, with an input of 64x64, I tried this as well, but found that five layers gave more accuracy using my specific setup and hyperparameters. Patch-PnP still achieves much better accuracy than my experimental model, the reason for which is not entirely clear to me but will be discussed in the discussion chapter.

The specific design I decided upon was using five convolutional layers, each followed by a batch normalization layer and 2x2 max-pooling similar to Patch-PnP. To combat overfitting I used a 2D DropBlock[6] layer before each convolution. A DropBlock layer is similar to a dropout layer, but is better suited for convolutions since it cancels contiguous 2D blocks of pixels instead of randomly spread individual pixels. This is better at restricting the flow of information through convolutional layers since the whole field of view of some neurons is cancelled[6]. GDR-Net does not use dropout in their Patch-PnP module. The reason it may be more important with dropout in my pose regression model is because I in some tests use pre-rendered query or coordinate inputs, which will not be augmented. Similar to Patch-PnP I use three separate heads to predict t_z, t_{xy} and R.

7.1.1 Pose Representation and Loss

For the pose representations I chose to implement the 6D rotation representation and SITE used by GDR-Net. These are the same as used by GDR-Nets successor GDRNPP which is currently the state of the art.

The loss functions I implemented are slightly different. For rotation I used the Frobenius distance between the ground truth and the predicted rotation matrix. I chose to use the Frobenius norm because it is easy to implement when the two rotations are given as matrices. I did a simple test with calculating the deviation of transformed points instead, similar to GDR-Net, and found no improvement in accuracy over using the Frobenius norm. For translation I use $\boldsymbol{L_1}$ distance for z and (x, y) separately, similar to GDR-Net.

7.2 Preliminary Tests

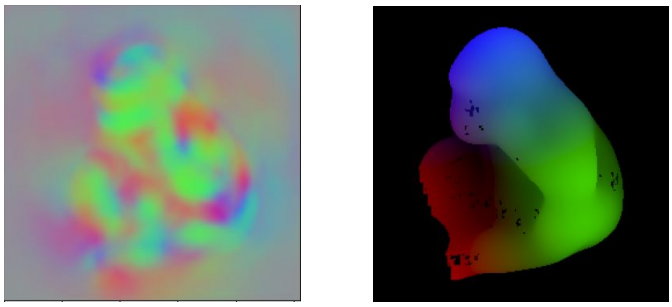
7.2.1 Sanity Check

First I needed to check if regressing a pose from the query image had any merit at all. To get high quality query images I downloaded a pre-trained model from SurfEmbs GitHub. With this model I could loop through the LMO RGB training images and generate a set of query images. In this way I created a new training set consisting of thousands of query images. Then, using my model, I compared regressing a pose from the query image versus regressing a pose directly from the raw training image. The "raw

training image" means the zoomed in crop taken from the RGB images in the training dataset. The purpose of this experiment was to see if the query image at the very least is a simpler representation to learn from than the raw image. To help the network understand where the object is it was given the object mask as well. Since the query image essentially contain the information of the object mask, as seen in figure 7.1, I saw it as more fair to at least give the model trained on raw RGB images the object mask as well. The expectation is that regression from the raw image is impossible without an encoder-decoder. The results showed that the network was not able to learn anything meaningful when inputted with the raw images, whereas it learned quickly with the query image and achieved reasonable results. This showed that having an encoder-decoder transform the raw image into a query image is helpful for the pose regression network. The next step is to compare it with a dense correspondence image like M_{3D-2D} on its efficacy as an intermediate representation.

7.2.2 Comparison With Coordinate Map

Due to limited time I chose not to train an encoder-decoder to regress object coordinates, but rather generated the ground truth coordinates by using a renderer in SurfEmbs source code. The renderer takes as input the pose of an object and renders it such that the computer can calculate the object points present at each pixel, shown in figure 7.1. The output of the renderer is the ground truth of what a coordinate regression network would output. Therefore the following tests are done on unrealistically good coordinates. The model was also tested on images generated by the renderer. Training the pose regression network on the coordinate maps gave approximately the same accuracy as the query images, which indicates that the query image may be able to replace it without affecting accuracy. There was too much variance in training and testing to say whether one or the other was more accurate. If what the results indicate is true, then the next step is to check if the exclusion of M_{SRA} affects accuracy.



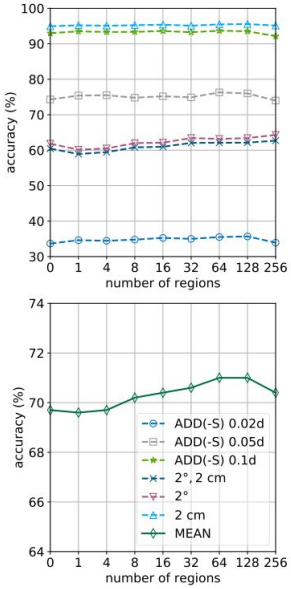
(a) (b)

Figure 7.1: (a): An RGB visualization of a premade query image of the gorilla object. Query images are in actuality 12 dimensional, so the dimension is reduced to 3 for visualization by summing together groups of four channels. (b): A rendered coordinate image of the gorilla object. Both images are formed from the same training image. In instances where the object is occluded the rendered coordinate image is altered to only show the visible parts of the object. Query images are formed from the actual images and so already contain occlusions. The rendered image in this figure has some black artefacts for unknown reasons.

7.2.3 Exclusion of M_{SRA}

What I was interested in testing is if M_{SRA} increases accuracy outside of its symmetric guidance. Can it for example increase accuracy on unsymmetrical objects? If it does not increase accuracy outside of its symmetric guidance then it should be safe to exclude since the query image already should cover the issue with symmetries.

Due to limited time I could not implement furthest point sampling and train a network to regress M_{SRA}. Instead I looked at what the authors of GDR-Net state is the benefit of M_{SRA}. They never state that pose accuracy was a reason for its inclusion. They state that it was included to give "ambiguity-aware supervision" and that M_{SRA} "implicitly represents the symmetries of an object". Secondly they state that M_{SRA} acts as an auxiliary task on top of M_{3D}, which "eases the learning of M_{3D} by first locating coarse regions and then regressing finer coordinates". For unsymmetrical objects it therefore seems M_{SRA} is only beneficial for increasing the accuracy of M_{3D}, which in turn increases the accuracy of the pose. Therefore M_{SRA} does not necessarily directly affect the final pose of unsymmetrical objects. Since my tests on coordinate maps versus query images used the ground truth coordinate maps, there is reason to believe the addition of M_{SRA} would not increase pose accuracy in those tests. Of course ground truth coordinates cannot be ambiguous in terms of symmetry, and so M_{SRA} should have no effect on any object, symmetric or



Source: Taken from [21]

Figure 7.2: Results from ablation studies as published in the GDR-Net paper[21]. The lines show what percentage of the test images have an error under certain rotation and translation thresholds. ADD(-S) is a metric that checks if the average error of the transformed object points is less than 10% of the objects diameter(0.1d). The bottom graph shows the mean improvement in accuracy across the metrics.

asymmetric, in my tests.

Additionally, the GDR-Net authors did some ablation studies on the number of fragments in M_{SRA}. The studies are done on the LM dataset, which contains limited symmetries. The results are shown in figure 7.2. The results show approximately a 1% increase in accuracy when using 64 fragments as opposed to 0 fragments. This shows a limited increase in accuracy on datasets with little symmetry. This indicates that by excluding the M_{SRA} image and instead using the symmetry aware query image, there is likely no decrease in accuracy.

7.3 Combination of Patch-PnP and SurfEmb

So far my model was fed premade query images from a separate query model without there being any connection between the two models. This means the query model has no vision of what the final and most important

output should be, namely a pose. Instead, the query model is trained only to produce good queries. If the models instead were connected such that the gradients from the pose loss can be traced all the way back to the input image, the query model may learn to produce a slightly different query image better suited for a model to regress a pose from. This is an important concept in deep learning and pose regression and is discussed a lot in the literature. For example [2] created a differentiable RANSAC procedure with the purpose of back-propagating the final loss through the RANSAC procedure and to the preceding model. The authors of [2] state that "Part of this recent success (in deep learning) is the ability to perform end-to-end training, i.e. propagating gradients back through an entire pipeline to allow the direct optimization of a task-specific loss function". Clearly an important concept in deep learning is to give the model a view of what the final output should be, not just what some intermediate representation should be. The intermediate representation should be there to guide rather than to dictate as it is probably not the most optimal representation, only a human approximation.

The simplest way of implementing the combined model was to use SurfEmbs source code and modify their query-key model class. SurfEmbs model class has a step function which is responsible for performing the forward calculation and the loss calculation. I added my pose regression model to the class and modified the forward function to also regress a pose. The step function of the combined model is found in the appendix. The query images calculated in the forward call are now also fed into a pose regression model and the pose loss is calculated. The final loss is the sum of the original SurfEmb embedding loss calculated from the query images, and the pose losses calculated from the pose prediction.

$$\boldsymbol{L}_{tot} = \boldsymbol{L}_{nce} + \boldsymbol{L}_{mask} + \boldsymbol{L}_{z} + \boldsymbol{L}_{center} + \boldsymbol{L}_{R} \tag{7.1}$$

The first two loss terms are the infoNCE and mask loss from SurfEmb, and the final three terms are pose losses as implemented by me. I added each of the models parameters to the same optimizer object so that the model could be trained end to end as one single unified model. The model was then trained with a training loop and the Adam optimizer as implemented in SurfEmb's source code.

7.4 Overview of the Tested Models

With the preliminary testing and implementation completed I decided to test four different models. I give them the following names for convenience:

- $\mathcal{M}_{pre.q}$ - A pretrained query model is downloaded from SurfEmbs GitHub and its output is fed into a pose regression model.
- $\mathcal{M}_{pre.c}$ - Ground truth coordinate maps are generated and fed into a pose regression model.
- $\mathcal{M}_{e2e}^{all}$ - The query and pose regression models are connected and trained end-to-end with pose losses and SurfEmbs embedding losses.
- $\mathcal{M}_{e2e}^{pose}$ - The query and pose regression models are connected and trained end-to-end with only pose losses.

All four methods use the same pose regression model, they are only distinguished by what happens before the pose regression and what losses are used. All four methods are also given a 2D positional encoding map, as covered in the GDR-Net chapter, as it is shown to increase accuracy.

Testing these four different models is a simple ablation study that should give some insight into the effectiveness of queries versus coordinates and the benefits of end-to-end training. Looking at $\mathcal{M}_{pre.q}$ versus $\mathcal{M}_{pre.c}$ will give insight into if coordinates, like used in GDR-Net, are better than queries as an intermediate representation. Testing this internally using the same model, data augmentation and training scheme means the results are comparable. Though it is not a perfect comparison since the coordinates used in my experiments are generated by a renderer and are therefore the ground truth coordinates, which are unrealistically good. Comparing the two end-to-end models should give insight into if using queries as an intermediate representation is better than having no intermediate representation at all.

I train a separate model for each object. Usually methods on the BOP website train a separate model for each object. SurfEmb chooses to have a common encoder, but has different decoders for each object. GDR-Net experiments with using both one common and spearate models, and find unsurprisingly that using one model per object performs best.

7.5 Training and Test Data

I chose to train and test on the LMO dataset. This dataset only has syntehtic training images available. These are created by software and can be downloaded from the BOP website. An example of such an image is given in figure 7.3. I downloaded all 50.000 synthetic images which each contains some or all of the dataset's objects. This is a simple dataset

to test on since there is little symmetry, meaning I don't have to implement a symmetry aware loss for rotation. Implementing symmetry aware rotation loss like in GDR-Net seemed to be cumbersome and time consuming since all symmetries for all objects would need to be manually found. If I was to implement it, I would not be able to compare it to GDR-Nets symmetry map M_{SRA} internally without implementing that as well, which I found to be too time consuming. I instead decided to use the time to try and improve the accuracy on non-symmetric objects. Even though the potential benefit of the queries lies in their symmetry awareness, their accuracy on non-symmetrical objects is still very interesting. The non-symmetrical accuracy will, to the best of my knowledge, carry over to symmetric objects when a symmetry aware loss function is used. There is still the chance that the query representation performs worse than GDR-Net for non-symmetrical objects, but equal or better for highly symmetrical objects, and so further work should test this.

Figure 7.3: An example of what a synthetic image from the LMO dataset looks like. The image has dimensions 640x480. A red circle is added to highlight the glue, and is not a part of the original image. The glue is displayed by 1050 pixels and has a visible fraction of 100%. Most methods use a separate object detector to zoom in on the object of interest during inference. In this image the gorilla figurine(red), the cat object(pink behind the screwdriver) and the stapler object(blue) are shown. These will be used for testing.

The three objects used for testing are the gorilla, cat and stapler objects and can be seen in figure 7.3. The objects were chosen arbitrarily.

Per object I used approximately 20.000-30.000 images for training and 500-700 for testing. Models are only trained and tested on images where over 1024 pixels and over 10% of the target object is visible. This is not necessarily added because of a limitation of the models. These are the same minimum visibility conditions used by SurfEmb when they trained their query model. Since I download a pretrained query model for exper-

imenting I decided to use the same conditions for my own models for a fairer comparison. I decided to also test them on the very same visibility conditions. If the model was to be used in conditions where the visibility often is poorer than this, one should consider lowering these minimums for training. Figure 7.3 gives an idea of what the minimum pixel count means in practice. Some objects had more images fulfilling the minimum visibility criteria available, like the stapler and cat object with around 30.000 each, whereas the gorilla had fewer, around 20.000.

7.5.1 Data Augmentation

I use the same data augmentation as SurfEmb. This includes some augmentations from the Albumentations library: GaussianBlur, ISONOise, GaussNoise, CLAHE, ColorJitter and CoarseDropout. SurfEmb also uses some self-implemented transforms called DebayerArtefacts and Unsharpen. Using the same augmentations means it is easier to compare my method with SurfEmb. These augmentations are only used in my end-to-end models, not the ones using pre-made inputs. The reason for this is that the augmentations were meant to be used on the input RGB images, not intermediate representations. This means the models with premade inputs have less variation in training images. To compensate for this I use DropBlock layers when training the pre-rendered models to combat overfitting.

7.5.2 Optimizers and Hyperparameters

The Ranger optimizer is used when training the models with pre-rendered input, which is the same optimizer used by GDR-Net. The Ranger optimizer is a combination of Rectified Adam (RAdam)[16], Lookahead[23] and Gradient Centralization[22]. When training end-to-end models, the Adam optimizer is used as it was already used in the SurfEmb code. Since the pose regression model was appended to the SurfEmb model, and trained jointly, it was simplest to have a common optimizer. Different optimizers were used because the pre-rendered models were created before I had the idea to make the end-to-end models. The potential impact of this difference will be discussed in the discussion chapter. As further work the optimizers should be investigated further, and perhaps switched or modified. The learning rate is set to be 3×10^{-5} in both models, and batch size is set to 16.

7.5.3 Test Metrics

For ease of implementation, the Frobenius norm is used to compare the difference between the predicted rotation matrix and the ground truth rotation matrix. The Frobenius norm of the difference between two matrices

has the following mathematical expression:

$$||A - B||_F = \sqrt{\sum_{i=0}^{n}\sum_{j=0}^{m}(a_{ij} - b_{ij})^2} \tag{7.2}$$

Where A and B are two matrices of dimension $m \times n$ containing elements a_{ij} and b_{ij} respectively. In tests on real data the angle between the two matrices are also listed, as I later found it to be a simple calculation. To get an understanding of the relation between the Frobenius norm and the angle it is recommended to check table 8.4 in the real data results.

For translation the L_2 norm is used, meaning it is the conventional distance between the predicted and the ground truth translation.

Chapter 8

Results

8.1 $\mathcal{M}_{pre.q}$ vs $\mathcal{M}_{pre.c}$

Both models were trained for 20 epochs on 20.000 synthetic images of the gorilla object, and tested on a synthetic dataset consisting of 417 images. The average frobenius distance between the ground truth and predicted rotation matrices is listed. For reference, randomly guessing rotation matrices yields on average around a 2.6 Frobenius error. Objects are usually positioned between 600-1200 millimeters from the camera. The L_2 distance is used to calculate error in translation.

	$\boldsymbol{R}_{err}$	$\boldsymbol{t}_{err}(mm)$
$\mathcal{M}_{pre.q}$	0.3417	27.06
$\mathcal{M}_{pre.c}$	0.3330	25.84

8.2 $\mathcal{M}_{e2e}^{all}$ vs $\mathcal{M}_{e2e}^{pose}$

In the tables below the rotation error is measured as the Frobenius norm of the difference between the ground truth rotation matrix and the predicted rotation matrix.

	$\boldsymbol{R}_{err}$	$\boldsymbol{t}_{err}(mm)$
$\mathcal{M}_{e2e}^{all}$	0.3054	23.47
$\mathcal{M}_{e2e}^{pose}$	0.3076	31.77

Table 8.1: Average rotation and translation error on the gorilla object. Trained for 20 epochs on 20.657 images. Tested on 417 synthetic images.

	$\boldsymbol{R}_{err}$	$\boldsymbol{t}_{err}(mm)$
$\mathcal{M}_{e2e}^{all}$	0.2451	23.62
$\mathcal{M}_{e2e}^{pose}$	0.2652	28.46

Table 8.2: Average rotation and translation error on the cat object. Trained for 20 epochs on 32.612 images. Tested on 698 synthetic images.

	$\boldsymbol{R}_{err}$	$\boldsymbol{t}_{err}(mm)$
$\mathcal{M}_{e2e}^{all}$	0.2571	31.16
$\mathcal{M}_{e2e}^{pose}$	0.2807	32.84

Table 8.3: Average rotation and translation error on the stapler object. Trained for 20 epochs on 35121 images. Tested on 689 synthetic images.

8.3 Comparison With SurfEmb on Real Data

The BOP website has made available the real images used for testing in previous years. SurfEmb's GitHub provides some detector bounding boxes for these images, making it easy to test my models on them. The following test was conducted on all 133 images of the cat object using an end-to-end model with all losses. SurfEmb was run on the same images. The average rotation and translation error are given in table 8.4.

	$\boldsymbol{R}(\circ)$	$\boldsymbol{R}$(Frobenius)	$\boldsymbol{t}(mm)$
SurfEmb	6.0	0.150	26.39
$\mathcal{M}_{e2e}^{all}$	19.6	0.478	49.40

Table 8.4: Mean rotation and translation error when tested on 133 real images of the cat object. Rotation error is also given in degrees as they are easier to understand.

8.4 Runtime

The timing tests were done using an NVIDIA A100 GPU and an Intel(R) Xeon(R) Gold 6248R CPU. My end-to-end method uses mostly the GPU, while SurfEmb uses the GPU and the CPU. SurfEmb uses over 95% of its runtime on the RANSAC procedure, which runs on the CPU.

My end-to-end model using queries has a runtime of 25ms. Running SurfEmb on the same GPU and CPU uses 380ms for estimation and 120ms for their refinement process, totalling 500ms. This means my proposed method uses only 5% of the runtime, meaning it could be used in a real-time system. A time of 25ms per estimation gives 40 frames per second, which is more than enough for many robots and systems.

Chapter 9

Discussion

9.1 Pre-Made Coordinate and Query Models

$\mathcal{M}_{pre.c}$ has a small edge in accuracy in both rotation and translation. This may be due to the fact that the generated coordinate maps are ground truth maps, whereas the query image is formed by a model with some inaccuracy. I believe the results could be improved for both versions with better training hyper-parameters and model architecture, since the results are not as good as the ones achieved by GDR-Net. With fine-tuning it is possible the coordinate based method pulls further ahead, but this would need to be tested. Nevertheless, the results show that it is possible to regress a pose directly from the query image and produce reasonable results, which was not a given.

If the queries also give the same accuracy as coordinate maps for better tuned models, then they may serve as a more compact intermediate representation than GDR-Net's, using only 12 channels as opposed to 64.

9.2 End-to-End Models

Both end-to-end versions trained on the gorilla object perform better than the ones using a pre-rendered coordinates or queries, except for the translation error of $\mathcal{M}_{e2e}^{pose}$ which was the worst of the four. It is the only model without any intermediate representation, and so this could be why translation estimation was more difficult. The results indicate that the end-to-end model with queries as an intermediate representation performs better than the one without any intermediate representation. This shows both that end-to-end training is beneficial and that queries are beneficial as

an intermediate representation. Some of the reason queries are beneficial could be because it acts as an object mask, which helps teach the network to only pay attention to the object of interest. I believe the re-coloration that queries essentially do is also helpful, especially for textureless objects like the gorilla object.

The end-to-end models use the Adam optimizer whereas the pre-rendered models use the Ranger optimizer. The Ranger optimizer is an extension of Adam and should in theory perform better.

Looking at the accuracy on the gorilla object versus the cat and stapler objet, it is seen that the rotation error decreases as the number of training images increases, which is not suprising.

9.3 Comparison With SurfEmb on Real Data

My models error doubled on both rotation and translation when tested on real data. This could mean the augmentations should be amplified more. The results show that SurfEmb has a very clear edge in accuracy on real data, especially on rotation. I was not able to reproduce the same accuracy as GDR-Net, which is similar to SurfEmb in accuracy. GDR-Net had 95.5% of translation results withing 2cm, and 63.2% of rotations within 2 degrees on the LM dataset. The LM dataset has less occlusion than LMO and so it is easier, but since my model fails to achieve the same accuracy on un-occluded instances within LMO, my model is likely not as accurate. Interestingly, SurfEmb also performs worse with a 6 degree average rotation error, which indicates that the test data was more difficult than the one GDR-Net tested on. My model does best on translation, where the average error is less than 5cm. It seems rotation is more difficult to learn, which makes sense since it is a 6 dimensional prediction with the 6D representation discussed in the theory section, whereas the disentangled translation is only 1 and 2 dimensions.

Both GDR-Net and SurfEmb are trained on the same training images I used, and tested on the same test images. The data can therefore not be the limiting factor. The augmentations used are the same as used by SurfEmb. I suspect my pose regression model either has a sub-optimal architecture or was trained with sub-optimal hyper-parameters. I trained only for 20 epochs which is a lot less than GDR-Net, which trained for 160 epochs. I trained for 20 epochs because the model did not improve beyond that point, which may indicate I had too large of a learning rate. I attempted to decrease the learning rate and train for more epochs, but due to problems with my sessions on IDUN shutting down for unknown reasons after approximately 12 hours, I could not train for more epochs.

End-to-end methods other than GDR-Net have also achieved similar accuracy to SurfEmb. SC6D[3] is an example of such a method, achieving the

same accuracy for the TLESS dataset. SC6D uses a similar architecture with an encoder-decoder followed by a pose decoder. They do not use any intermediate representation except for a mask prediction. Clearly it should be possible to get better accuracy without having the same intermediate representations as GDR-Net. I therefore believe the main limiting factors for my models accuracy is the architecture, training or some overlooked details, and not the usage of query images as intermediate representation. The fact that the end-to-end model with queries performed better than the one with no intermediate representations supports this. The fact that the pose regression model inputted with pre-rendered coordinates also has worse accuracy also indicates that the training or architecture is the main limiting factor.

My model however has a drastic improvement on runtime over SurfEmb, achieving 20 times faster inference. This was expected as the RANSAC procedure used by SurfEmb does thousands of iterations on the CPU. The runtime is similar to GDR-Net which has a 7ms inference on a NVIDIA 2080Ti GPU. My runtime is likely larger than GDR-Net because of my models larger size, which probably could be reduced.

9.4 Possible Improvements for Better Accuracy

In deep learning it can be difficult to know which aspects impact performance the most. Below are ten aspects that should be considered when attempting to increase the accuracy of the presented method.

- **Potential bugs** - There may have been bugs in my code stemming from unintended use of SurfEmbs source code. For example an augmentation may have been used for something it was not meant to, or an optimizer is not functioning properly. I have no specific reason to believe this, but I would not count it out. My method could be coded from scratch to have more control over what is happening.
- **Learning rate** - My end-to-end models were observed to stop improvement, even on the training loss, after 15-30 epochs. This could mean the learning rate is too high, making it difficult for the network to converge.
- **Epochs** - The model should be trained over more epochs with a lower learning rate.
- **Data augmentation** - My model doubles in error when testing on real data. Even though SurfEmb uses the same augmentations, my model relies more on deep learning and so might require stronger or different augmentations. With stronger augmentations the model could become more robust.

- **Filter and image size** - I used a 7x7 filter followed by four 3x3 filters in the pose regression network. Since the images it takes as input have dimensions 256x256, the filters have a relatively smaller size than GDR-Net's, which uses 3x3 filters on 64x64 images. I deviated from GDR-Net on image size and filters because in my testing it gave better results, but this could change when other aspects like learning rate and number of epochs are changed.
- **Dropout** - I observed, especially when training the models with pre-rendered inputs, that they were prone to overfitting after about 10 epochs. This should not happen after so few epochs. This happened even though there were DropBlock layers before each convolution with 0.4-0.5 drop chance, meaning approximately half of pixels were cancelled out. More dropout could be used. There are also other methods of decreasing overfitting, like reducing model complexity by having fewer convolutional layers.
- **Model complexity** - My model uses 5 convolutional layers since it increased accuracy in my tests, but this may change when trained for more epochs. Since the model starts overfitting, and especially since GDR-Net only uses three convolutional layers, a model with less layers should be tested. It is possible my larger model seems to be more accurate only because it learns faster, and that a smaller model will outperform it when trained for longer.
- **Loss function** - I use the Frobenius norm when calculating rotation error, whereas GDR-Net uses the deviation of object points when transformed with the predicted transformation versus the ground truth transformation. I use the Frobenius norm because it is the same metric used when comparing my models, and it is simple to implement. Small initial tests indicated that the Frobenius norm performed equally to transforming points, but this should be tested more thoroughly. If for example the Frobenius norm has a too steep gradient around zero it may be harder for the model to converge reliably, or if it is too flat convergence can be too slow.
- **Optimizer** - The optimizer used is the Adam optimizer, which is not straightforward to understand. I used the same parameters as SurfEmb but cannot say for certain that the optimizer is working as intended. The training log seemed to indicate that the learning rate never changed during training, but I do not know enough of the specifics of the optimizer or how SurfEmb logs the learning rate to know if this is true or intended. A more thorough review of the optimizer should be done because having an adaptive learning rate can minimize overshoot and make convergence easier.
- **Weighting of the loss terms** - The five loss terms in my model, shown in equation 7.1, are each weighted by a scalar. The weight determines how much the model prioritizes them during training.

My weights were within a factor of 10 of each other. More extreme weights could be tested.

These are the ten most obvious things that can be experimented with. There are probably more aspects that more experienced deep learning engineers can identify.

Chapter 10

Further Work

The models I present should be tested on truly symmetric objects, such as in the TLESS dataset. A symmetry aware loss function would need to be implemented to do so. Other than this, for the most part, further work should be aimed at refining the model and training as suggested in the discussion section above.

There may have been bugs in my code stemming from unintended use of SurfEmbs source code. As further work one should also strongly consider coding from scratch.

Chapter 11

Conclusion

This work shows that a Convolutional Neural Network can be trained to regress poses from symmetry aware query images as presented by SurfEmb, giving a drastic increase in speed compared to SurfEmb, but with lower accuracy. To the best of my knowledge this has not been attempted before. Whether these queries can replace dense correspondences as intermediate representation without decreasing accuracy is not entirely certain without testing on better tuned models. The query image is shown to increase the accuracy when used as an intermediate representation in end-to-end models, as opposed to not having any intermediate representation. The exact reason for why the presented method is not as accurate as GDR-Net is not certain, but there is reason to believe that the accuracy will improve with longer training, better training- and optimizer parameters, and better model architecture.

Bibliography

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33833-8.

[2] Eric Brachmann et al. "DSAC — Differentiable RANSAC for Camera Localization". In: July 2017, pp. 2492–2500. DOI: 10.1109/CVPR.2017.267.

[3] Dingding Cai, Janne Heikkilä, and Esa Rahtu. *SC6D: Symmetry-agnostic and Correspondence-free 6D Object Pose Estimation*. 2022. arXiv: 2208.02129 [cs.CV].

[4] 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.

[5] 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.

[6] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. "DropBlock: A regularization method for convolutional networks". In: *CoRR* abs/1810.12890 (2018). arXiv: 1810.12890. URL: http://arxiv.org/abs/1810.12890.

[7] 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.

[8] 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.

[9] 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.

[10] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[11] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (Dec. 2014).

[12] 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.

[13] 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.

[14] Shiqi Li, Chi Xu, and Ming Xie. “A Robust O(n) Solution to the Perspective-n-Point Problem”. In: *IEEE transactions on pattern analysis and machine intelligence* 34 (Jan. 2012). DOI: 10.1109/TPAMI.2012.41.

[15] Zhigang Li, Gu Wang, and Xiangyang Ji. “CDPN: Coordinates-Based Disentangled Pose Network for Real-Time RGB-Based 6-DoF Object Pose Estimation”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 7677–7686. DOI: 10.1109/ICCV.2019.00777.

[16] Liyuan Liu et al. “On the Variance of the Adaptive Learning Rate and Beyond”. In: *CoRR* abs/1908.03265 (2019). arXiv: 1908.03265. URL: http://arxiv.org/abs/1908.03265.

[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] 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.

[19] 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.

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

[21] Gu Wang et al. “GDR-Net: Geometry-Guided Direct Regression Network for Monocular 6D Object Pose Estimation”. In: *CoRR* abs/2102.12145 (2021). arXiv: 2102.12145. URL: https://arxiv.org/abs/2102.12145.

[22] Hongwei Yong et al. “Gradient Centralization: A New Optimization Technique for Deep Neural Networks”. In: *CoRR* abs/2004.01461 (2020). arXiv: 2004.01461. URL: https://arxiv.org/abs/2004.01461.

[23] Michael R. Zhang et al. “Lookahead Optimizer: k steps forward, 1 step back”. In: *CoRR* abs/1907.08610 (2019). arXiv: 1907.08610. URL: http://arxiv.org/abs/1907.08610.

[24] Yinqiang Zheng et al. “Revisiting the PnP Problem: A Fast, General and Optimal Solution”. In: *2013 IEEE International Conference on Computer Vision*. 2013, pp. 2344–2351. DOI: 10.1109/ICCV.2013.291.

[25] Yi Zhou et al. “On the Continuity of Rotation Representations in Neural Networks”. In: *CoRR* abs/1812.07035 (2018). arXiv: 1812.07035. URL: http://arxiv.org/abs/1812.07035.

Appendix A

The step function of the combined end-to-end model and my pose regression model. The appended step function is a modification of SurfEmbs source code. The original step function is found in SurfEmbs surface_embedding.py file on their GitHub: (https://github.com/rasmushaugaard/surfemb).

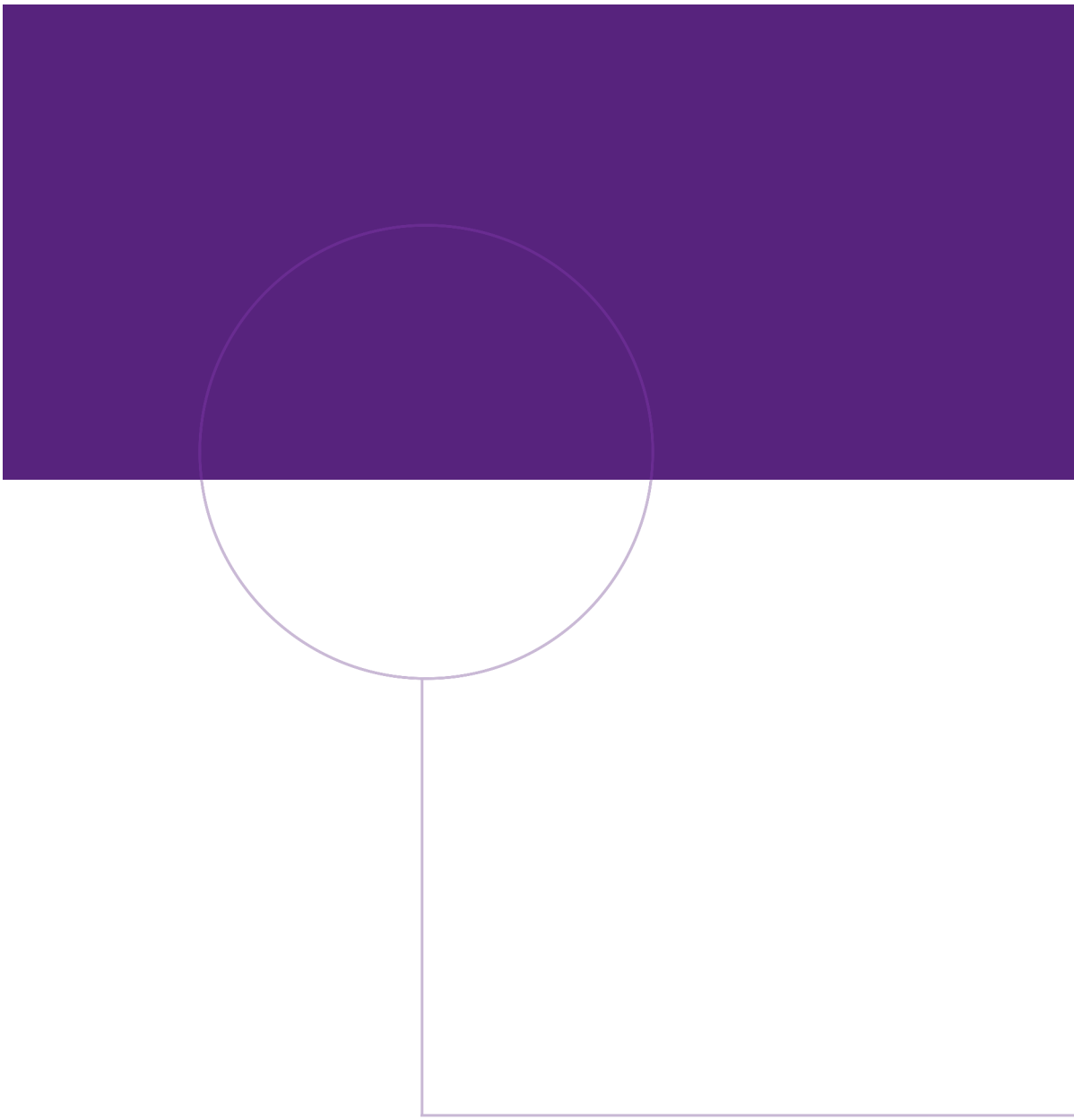
In [ ]:

```
def step(self, batch, log_prefix):
        img = batch['rgb_crop']  # (B, 3, H, W)
        if log_prefix=='test':  #Used when testing on synthetic data
            img = batch['rgb_crop_test'] #Just a normal RGB crop with manually added normalization which the pretrained resnet18 cnn expects.
                                         #Normalization happens in the dataloader for normal RGB crops during training

        coord_img = batch['obj_coord'].to(0)  # (B, H, W, 4) [-1, 1]
        obj_idx = batch['obj_idx']  # (B,)
        coords_neg = batch['surface_samples'].to(0) # (B, n_neg, 3) [-1, 1]
        mask_samples = batch['mask_samples'].to(0)  # (B, n_pos, 2)

        device = img.device
        B, _, H, W = img.shape
        assert coords_neg.shape[1] == self.n_neg
        mask = coord_img[..., 3] == 1.  # (B, H, W)
        y, x = mask_samples.permute(2, 0, 1)  # 2 x (B, n_pos)

        if self.separate_decoders:
            with timer("cnn",False):
                cnn_out = self.cnn(img.to(0), obj_idx)  # (B, 1 + emb_dim, H, W)
                mask_lgts = cnn_out[:, 0]  # (B, H, W)
                queries_img = cnn_out[:, 1:]  # (B, emb_dim, H, W)
        else:
            cnn_out = self.cnn(img)  # (B, n_objs + n_objs * emb_dim, H, W)
            mask_lgts = cnn_out[torch.arange(B), obj_idx]  # (B, H, W)
            queries = cnn_out[:, self.n_objs:].view(B, self.n_objs, self.emb_dim, H, W)
            queries_img = queries[torch.arange(B), obj_idx]  # (B, emb_dim, H, W)

        mask_prob = torch.sigmoid(mask_lgts)  # (B, H, W)
        mask_loss = F.binary_cross_entropy(mask_prob, mask.type_as(mask_prob))

        queries = queries_img[torch.arange(B).view(B, 1), :, y, x]  # (B, n_pos, emb_dim)

        # compute similarities for positive pairs
        coords_pos = coord_img[torch.arange(B).view(B, 1), y, x, :3]  # (B, n_pos, 3) [-1, 1]
        coords_pos += torch.randn_like(coords_pos) * self.key_noise
        keys_pos = torch.stack([self.mlps[i](c) for i, c in zip(obj_idx, coords_pos)])  # (B, n_pos, emb_dim)
        sim_pos = (queries * keys_pos).sum(dim=-1, keepdim=True)  # (B, n_pos, 1)

        # compute similarities for negative pairs
        coords_neg += torch.randn_like(coords_neg) * self.key_noise
        keys_neg = torch.stack([self.mlps[i](v) for i, v in zip(obj_idx, coords_neg)])  # (B, n_neg, n_dim)
        sim_neg = queries @ keys_neg.permute(0, 2, 1)  # (B, n_pos, n_neg)

        #Running pose estimation model
        with timer("allPatchPnP",False):
            for key, value in batch.items():  #Moving tensors to cpu to be used in calculations
                if key == 'AABB_crop' or key == 'bbox_visib' or key=='bbox_obj' : continue #These are numpy arrays already on cpu
                batch[key] = batch[key].to(device='cpu')

            with timer("patchpnp", False):
                q = torch.cat((queries_img.to(0),self.pos_encoding.to(0)), dim=1).to(device=device)
                dtxy, dtz, r6d = self.PnP_model(q)
                dtxy, dtz, r6d = dtxy.to(device='cpu'), dtz.to(device='cpu'), r6d.to(device='cpu')
            gt_R_b = batch['cam_R_obj'].to(device='cpu')
            gt_t_b = batch['cam_t_obj'].to(device='cpu')

            resize = queries_img.size()[2]
            batch_size = img.size()[0]
            s_zoom = resize
            loss_txy = 0
            loss_tz = 0
            loss_R = 0
            loss_R_frob = 0
            tz_diff = 0
            dist_t=0

            for b in range(batch_size):

                #Defining variables for use in SITE calculation
                crop = [_[b] for _ in batch['AABB_crop']]
                w, h = crop[2]-crop[0], crop[3]-crop[1]
                w = w.to(device='cpu')
                h = h.to(device='cpu')
                cx, cy = int(crop[0] + w/2), int(crop[1] + h/2)
                s0 = max(w, h).to(device='cpu')
                r = s_zoom/s0
                batch['K'] = batch['K'].to(device='cpu')

                #Finding ground truth SITE parameters
                o_gt = batch['K'][b]@gt_t_b[b]
                o_gt = o_gt/o_gt[2]
                t_gt = gt_t_b[b]
                dtx_gt = (o_gt[0]-cx)/w
                dty_gt = (o_gt[1]-cy)/h
                dtz_gt = t_gt[2]/r

                #Constructing predicted t
                tz_pred = r*dtz[b]
                ox_pred = w*dtxy[b][0] + cx
                oy_pred = h*dtxy[b][1] + cy
                t_pred = torch.inverse(batch['K'][b]).double()*tz_pred@torch.Tensor([ox_pred,oy_pred,1]).double()
                t_pred = t_pred.detach().numpy()
                dist_t += np.linalg.norm(t_pred-t_gt.numpy()[:,0])

                #Constructing predicted R
                R_gt = torch.Tensor(gt_R_b[b]).double()
                r1, r2 = r6d[b][:3], r6d[b][3:]
```

```
            R1 = F.normalize(r1,dim=0)
            R3 = F.normalize(torch.cross(R1, r2), dim=0)
            R2 = torch.cross(R3,R1)
            R_pred = torch.stack((R1,R2,R3), dim=1)

            #Calculating losses and adding to batch total
            loss_tz += torch.abs(dtz[b]-dtz_gt)
            tz_diff += np.abs(t_pred[-1] - t_gt[-1])
            loss_txy += torch.abs(dtxy[b][0]-dtx_gt) + torch.abs(dtxy[b][1]-dty_gt)
            loss_R += torch.linalg.matrix_norm(R_gt - R_pred.double())
            loss_R_frob += torch.linalg.matrix_norm(R_gt - R_pred.double())

        #Calculating SurfEmbs losses
        lgts = torch.cat((sim_pos, sim_neg), dim=-1).permute(0, 2, 1)  # (B, 1 + n_neg, n_pos)
        target = torch.zeros(B, self.n_pos, device=device, dtype=torch.long)
        nce_loss = F.cross_entropy(lgts, target)
        loss = 5*loss_R.to(0) + loss_txy.to(0) + 0.01*loss_tz.to(0) + 10*mask_loss + 10*nce_loss
        return loss, loss_R_frob, dist_t
```

In [ ]:

```
import torch
from torch import nn
import torch.nn.functional as F
from dropblock import DropBlock2D

class ModelQueries(torch.nn.Module):
    def __init__(self,n_decoders=1):
        super(ModelQueries, self).__init__()
        self.decoders=nn.ModuleList([nn.ModuleDict({
        'bn-1' : nn.BatchNorm2d(14),
        'conv0' : nn.Conv2d(14,16,kernel_size=7,stride=1, padding=3),
        'bn0' : nn.BatchNorm2d(16),
        'conv1' :nn.Conv2d(16, 32, kernel_size=3,stride=1, padding=1),
        'bn1' : nn.BatchNorm2d(32),
        'conv2' : nn.Conv2d(32, 64, kernel_size=3,stride=1, padding=1),
        'bn2' : nn.BatchNorm2d(64),
        'conv3' : nn.Conv2d(64, 128, kernel_size=3,stride=1, padding=1),
        'bn3' : nn.BatchNorm2d(128),
        'conv4' : nn.Conv2d(128, 256, kernel_size=3,stride=1, padding=1),
        'bn4' : nn.BatchNorm2d(256),
        'fc1_r' : nn.Linear(256*8*8,1200),
        'fc1_z' : nn.Linear(256*8*8,1200),
        'fc1_xy' : nn.Linear(256*8*8,1200),
        'bn_xy1' : nn.BatchNorm1d(1200),
        'fc2_r' : nn.Linear(1200,256),
        'fc2_z' : nn.Linear(1200,256),
        'fc2_xy' : nn.Linear(1200,256),
        'bn_xy2' : nn.BatchNorm1d(256),
        'fc3_z' : nn.Linear(256,32),
        'fc4_z' : nn.Linear(32,1),
        'fc3_xy' : nn.Linear(256,2),
        'fc3_r' : nn.Linear(256,6)})
        for _ in range(n_decoders)])


    def forward(self, x, dec_idx=None, train=False):
        out_all_txy = []
        out_all_tz = []
        out_all_r = []
        dec_idx=0 #Force model to always use first decoder, as I train only on one object
        out = self.decoders[dec_idx]['bn-1'](x)
        out = DropBlock2D(block_size=20, drop_prob=0.5)(out)
        out = F.max_pool2d(F.relu(self.decoders[dec_idx]['conv0'](out)),2)
        out = DropBlock2D(block_size=10, drop_prob=0.5)(out)
        out=self.decoders[dec_idx]['bn0'](out)
        out = F.max_pool2d(F.relu(self.decoders[dec_idx]['bn1'](self.decoders[dec_idx]['conv1'](out))),2)
        out = DropBlock2D(block_size=5, drop_prob=0.4)(out)
        out = F.max_pool2d(F.relu(self.decoders[dec_idx]['conv2'](out)),2)
        out = DropBlock2D(block_size=1, drop_prob=0.4)(out)
        out = self.decoders[dec_idx]['bn2'](out)
        out = F.max_pool2d(F.leaky_relu(self.decoders[dec_idx]['bn3'](self.decoders[dec_idx]['conv3'](out))),2)
        out = F.max_pool2d(F.leaky_relu(self.decoders[dec_idx]['bn4'](self.decoders[dec_idx]['conv4'](out))),2)

        #Converting to 1D for use in MLP
        out_r = out.contiguous().view(out.size(0), -1)
        out_tz = out   #Branching translation predictions from rotation prediction
        out_txy = out

        out_r = F.relu(self.decoders[dec_idx]['fc1_r'](out_r))
        out_tz, out_txy = F.relu(self.decoders[dec_idx]['fc1_z'](out_tz)), F.relu(self.decoders[dec_idx]['fc1_xy'](out_txy))
        out_r = self.decoders[dec_idx]['fc1_dropout'](out_r)
        out_tz, out_txy = self.decoders[dec_idx]['fc1_dropout'](out_tz), self.decoders[dec_idx]['fc1_dropout'](out_txy)
        out_txy = self.decoders[dec_idx]['bn_xy1'](out_txy)
        out_r = F.relu(self.decoders[dec_idx]['fc2_r'](out_r))
        out_tz, out_txy = F.relu(self.decoders[dec_idx]['fc2_z'](out_tz)), F.relu(self.decoders[dec_idx]['fc2_xy'](out_txy))
        out_txy = self.decoders[dec_idx]['bn_xy2'](out_txy)

        out_tz = F.relu(self.decoders[dec_idx]['fc3_z'](out_tz))
        out_txy, out_tz = self.decoders[dec_idx]['fc3_xy'](out_txy), self.decoders[dec_idx]['fc4_z'](out_tz)
        out_r = self.decoders[dec_idx]['fc3_r'](out_r)
        out_all_txy.append(out_txy)
        out_all_tz.append(out_tz)
        out_all_r.append(out_r)
        return (torch.stack(out_all_txy)[0], torch.stack(out_all_tz)[0], torch.stack(out_all_r)[0])
```



NTNU

Norwegian University of
Science and Technology