To my family
ACKNOWLEDGEMENTS

I would like to acknowledge the State of Georgia and the Agricultural Technology Research Program at the Georgia Tech Research Institute for supporting the work described in this paper. I would like to express gratitude toward Benjamin Joffe. Benjamin has acted as a mentor throughout the development of this thesis. His guidance and insight has proved invaluable for the formulation of the problem investigated in this thesis as well as the experimental design and analysis. Working with him at the Food Processing Technology Division has strengthened my knowledge and appreciation of deep learning, robotics, and programming practices. I also owe great thanks to Dr. Zsolt Kira, who acted as my thesis advisor. His patience and constructive feedback has helped in the completion of this project.
# TABLE OF CONTENTS

Acknowledgments .............................................. v

List of Tables .................................................. viii

List of Figures .................................................. ix

Summary ........................................................ xii

Chapter 1: Introduction ..................................... 1

1.1 Motivation ................................................. 1

1.2 Pose Estimation ........................................... 1

1.3 Pose Estimation for Object Manipulation .............. 3

Chapter 2: Related Work .................................... 7

2.1 Pose estimation ............................................. 7

2.1.1 Instance Level Pose Estimation ....................... 7

2.1.2 Category-Level Pose Estimation .................... 11

2.2 Point Cloud Manipulation ............................... 14

2.2.1 Pointnets .............................................. 15

Chapter 3: Novel Approach to Category-Level Pose Estimation ........... 19

3.1 Instance-level Pose Estimation in the Wild ............ 19
3.2 Proposed Approach ......................................................... 21

Chapter 4: Experimental Design ........................................... 24
  4.1 Instance Level Method in Category Setting ....................... 24
  4.2 Canonicalization Experiments ...................................... 25
    4.2.1 Basic Pointnet Autoencoders ................................. 26
    4.2.2 Point Completion Network .................................. 27
    4.2.3 LOGAN: Latent Overcomplete Generative Adversarial Network . . 28
    4.2.4 3D Convolutional Autoencoder ............................. 29

Chapter 5: Results and Discussion ........................................ 31
  5.1 Densefusion Modification Verification Experiment ............. 31
  5.2 Canonicalization .................................................... 32
  5.3 Category-Level Pose Estimation via Canonicalization .......... 37

Chapter 6: Conclusion and Future Work ............................... 40

References ................................................................. 44
LIST OF TABLES

5.1 Densefusion Modification Validation .......................... 31
5.2 Canonicalization Experiments - Uniform SO(3) .................. 32
5.3 Canonicalization Experiments - Limited SO(3) .................. 32
5.4 Densefusion With and Without Canonicalization - Uniform SO(3) - ADD ≤2cm .......................... 38
5.5 Densefusion With and Without Canonicalization - Limited SO(3) - ADD ≤2cm .......................... 39
LIST OF FIGURES

1.1 An example of an object’s pose. The bottle has a local 3D coordinate frame which can be used to calculate transforms to and from other frames. [1] Image taken from [1]. .......................................................... 2

1.2 In Densefusion [2], the authors use pose estimation to enable the manipulation of objects .................................................. 4

1.3 In Joffe et. al [3], the authors use pose estimation to enable the manipulation of chicken products. The images on the left are of chickens in random starting orientations. Despite the fact that the chickens start in these random positions, the robotic system is able to infer their orientations using pose estimation, and re-orient the chicken to a canonical pose, as shown in the images on the right. .......................................................... 6

2.1 The different objects within the food category of the YCB dataset [4]. .... 8

2.2 The different objects within the tools category of the YCB dataset [4]. .... 8

2.3 An image of the architecture from PoseCNN [1]. This architecture regresses pose directly from RGB images. Image taken from [1]. ................. 10

2.4 The pipelined approach of PV-Net [11]. This architecture exploits the effectiveness of 2D convolutions to find 2D-3D correspondences. Given these correspondences, a PnP solver calculates the pose. Image taken from [11]. .. 10

2.5 Here are some images of some camera objects from the camera dataset [16]. Instead of having completely unique objects in several unique poses, this dataset contains several unique instances of a particular category. .......... 12

2.6 Architecture for the normalized object coordinate space network [16]. The 2D convolutional layers in this network map rgb pixels to positions within the NOCS coordinate system. The depth image is then used to project the NOCS representation to real-world positions. Image taken from [16]. .... 13
2.7 This figure shows the follow up work of NOCS [17]. Here, the authors discard the rgb information completely and rely on 3D processing to extract pose and articulation information from the scene. Image taken from [17].

2.8 Architecture of pointnet [18]. Image taken from [18].

2.9 Use cases of pointnet [18] on variably-sized, unordered pointclouds. Image taken from [18].

2.10 Architecture for point completion network [19]. This architecture consists of an autoencoder. The feature extractor for the encoder is a pointnet, while the decoder is a two-stage decoder. Stage one of the decoder is simply a series of fully connected layers that map the latent variable to a coarse-grained reconstruction. Stage two uses grid deformations inspired by folding-based decoders [21] to refine the output of the first stage.

2.11 Here are some example results from the LOGAN [22] network. The pointcloud on the far left show inputs of furniture. The images to the right show the translated pointclouds. They’ve been transformed from chair to tables and tables to chair in a differentiable way.

3.1 Flow of applying instance-level methods to category problems.

3.2 Flow of the proposed method.

4.1 This figure shows the original architecture of Densefusion.

4.2 This figure shows the modified architecture of Densefusion using only the depth processing.

4.3 This figure shows canonicalization working on some pointclouds of airplanes from the ShapeNet [24] dataset.

4.4 The 2D convolutional Unet [25]. The 3D autoencoder is identical, the only difference is the use of 3D convolutions as opposed to 2D. Image taken from [25].

5.1 Reconstructions of cars from the ShapeNet dataset [24]. The 3D Convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds are rotated with rotations taken from the limited SO(3) distribution.
5.2 Reconstructions of chairs from the ShapeNet dataset [24]. The 3D Convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds are rotated with rotations taken from the limited SO(3) distribution.

5.3 Attempted reconstructions of cars and chairs from the ShapeNet dataset [24]. The 3D convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds were rotated with rotations taken from the uniform SO(3) distribution. Here, the 3D convolutional autoencoder fails to reconstruct the pointcloud.
SUMMARY

This thesis describes a study of pose estimation. Pose estimation is the determination of an object’s position and orientation in some coordinate frame. Pose estimation has application in robotics, augmented/virtual reality, and human-computer interaction, among others. An overview of the field is given along with an explanation of some key challenges. First, this work specifically details instance-level pose estimation and its application. Instance-level pose estimation is the setting in which one specific instance of an object is used when querying pose. For example, a particular make, model, and year of a car may be chosen, and an instance-level method will predict the position and orientation of several different poses of that one particular car. Next, the thesis discusses category-level pose estimation and talks about some key differences between this and the instance-level setting. In the category-level setting, a method is responsible for predicting poses of several different instances of a particular category. Alluding to my earlier example using the car, in this setting, instead of predicting pose on one particular vehicle, a category-level algorithm will predict pose on any given object from an entire category, say sedan or pickup truck. This is a much tougher task, as now the algorithm must not only generalize to the various poses of the object, but it also must learn the general shape of the category. Although category-level methods handle different instances within a category better than instance-level methods, the bench-marked performance of these methods is not as high as their instance-level counterparts in terms of pose prediction accuracy. Finally, a novel idea for improving category-level pose estimation is presented along with some experiments showing this idea being applied. The idea is to reduce the category-level problem to a two stage process of transforming any given category instance to a canonical instance, which is a generic shape that represents all instances of a category, and then predicting the pose of the transformed instance. Here, this transformation of any given category instance to canonical instance is deemed canonicalization. Through some experiments, very slight improvement of pose estimation using canonical-
ization as a front-end preprocessing step over using an instance-level method alone applied in a category-level setting is shown. Although the canonicalization shows promise, when used in conjunction with the instance-level method, the improvement is slim.
CHAPTER 1
INTRODUCTION

1.1 Motivation

How is it that humans can so easily perceive, manipulate, and orient various objects? It seems as though no matter what object is presented before us, we are able to effectively see it and grasp it without much thought, regardless of its orientation. It is with this capability that humans are able to handle seemingly infinite objects, predict the motion of other objects, and visualize objects within their imagination. How can a robotic system possess this same capability, and what are the implications of this? If humanity wants robots to be truly useful outside of their restrictive, contrived lab settings, then these machines are going to need robust, flexible perception that is capable of understanding various objects and their associated poses in continuous 3D space.

1.2 Pose Estimation

The ability to discern the pose of an object relative to some arbitrary coordinate frame is pose estimation. In robotics problems, it is very common to define several coordinate frames of the various joints, links, and sensors of the system as well as the transformations between these coordinate frames. When performing certain tasks such as navigation or manipulation, it is often useful to express points present in one coordinate frame in terms of another coordinate frame. When expressing the pose of an object, it is necessary to first define a coordinate frame rigidly attached to this object. This allows transformations to be defined on this object. An example pose is shown in figure 1.1.
Figure 1.1: An example of an object’s pose. The bottle has a local 3D coordinate frame which can be used to calculate transforms to and from other frames. [1] Image taken from [1].

A transformation matrix can transform points, vectors, and entire coordinate frames from one coordinate system to the other. This matrix is comprised of a rotation component, and a translation component. An example transformation matrix in 3D is shown below in equation 1. The x, y, and z represent the translation component, and the $r_{11}...r_{33}$ show the rotation component.

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$  (1)

If a vision system can determine the pose of an object with respect to one of its camera coordinate frames, it has effectively calculated the transformation matrix from the camera to the object. This acquired knowledge of object pose opens up solutions to many problems in robotics and artificial intelligence. One such problem is that of robotic manipulation.
1.3 Pose Estimation for Object Manipulation

In order for robotic manipulators to effectively grasp and handle objects, they must have some understanding of how to grasp said object. This understanding can come in many forms. One possible way would be to compute grasp points on the object that robot should grasp. This would not require pose estimation, however. With knowledge about how to grasp an object, a robotic system can plan a trajectory to move the end-effector from its current configuration to its desired configuration. The problem of moving a robot’s end-effector from one configuration to another configuration, known as inverse kinematics, is more or less solved. So, if the robotic system could infer what configuration the end-effector needed to be in, then the system could move the end-effector from its current configuration to the desired configuration. If the target object is known to always be in the same configuration, then the target pose of the end-effector is constant. Say, for example, there is a robotic manipulator embedded within a manufacturing line. If objects coming down the assembly line are always going to be in the same position and orientation, then the problem of grasping the object reduces to moving the end-effector to the same pose, which aligns the gripper to the object to be grasped. If, however, the object were to be perturbed as it comes down the line and the position and orientation are different from what the robot expects, then when the robot attempts to grasp the object, it will likely fail. This is because the pose that the robot planned for in its actuation is not the same pose that the object is actually in. One could circumvent this problem by designing the assembly line such that upstream processes ensure that the object is in the exact same pose every time; however, this can be difficult for very large or small objects, and it can be difficult to readjust the pipeline when a new object must flow through the process.

A more flexible approach would detect the object and its pose in realtime, and adjust its manipulation planning accordingly. Pose estimation can enable this. When objects arrive from upstream processes, a pose estimation algorithm would detect the object’s pose
and relay that information to the manipulation system. With this information, the robotic system could move its manipulator accordingly to grasp the object. Wang et al [2] demonstrate a robotic system capable of grasping objects based on a pose calculation in a project called Densefusion. The authors first detect the object within the image. After acquiring a bounding box around the object, they use their pose estimation method to calculate the position and orientation of the object with respect to the robot. Once they have this pose estimate, they command the robot arm to place the end-effector in the proper pose and the robot grasps the object [2]. This is shown in figure 1.2, which is taken from [2].

Figure 1.2: In Densefusion [2], the authors use pose estimation to enable the manipulation of objects.

Another example of pose estimation in robotic manipulation is in this work by Joffe et al. [3]. Here, the authors use pose estimation to reorient a chicken to a certain pose. In this work, a chicken starts in a random pose that is initially unknown to the robot. After picking
up the chicken, the robot is able to orient the chicken in a canonical pose because it has an understanding of the starting pose. In fact, the authors do not use the pose information for grasping. They use the pose information to get the robot to orient the chicken in a consistent pose regardless of the starting configuration of the chicken. The object frame is attached to the kinematic chain of the manipulator, and the inverse kinematics solver is able to orient the chicken while it is in the gripper of the robot [3]. This is shown in figure 1.3 The key takeaway from these examples is: objects may appear in front of a robot in any given pose that is not known ahead of time, and with an adequate pose estimation algorithm, information about how the robot needs to place its end-effector can be calculated.
Figure 1.3: In Joffe et. al [3], the authors use pose estimation to enable the manipulation of chicken products. The images on the left are of chickens in random starting orientations. Despite the fact that the chickens start in these random positions, the robotic system is able to infer their orientations using pose estimation, and re-orient the chicken to a canonical pose, as shown in the images on the right.
CHAPTER 2
RELATED WORK

Now that the pose estimation problem has been defined and motivated with some example uses within robotic manipulation, an overview of the field will be given. The scope of this thesis is deep learning based methods for pose estimation and point cloud manipulation. This chapter will detail state-of-the-art methods in these areas.

2.1 Pose estimation

Like stated before, pose estimation is inferring an object’s coordinate frame transform with respect to some other coordinate frame. There have been many contributions to this problem, and this chapter will give an overview of the field. First, instance-level pose estimation will be covered, as this style of solving the problem is most common in the literature. Secondly, category-level pose estimation will be discussed.

2.1.1 Instance Level Pose Estimation

Instance level pose estimation trains deep neural networks to infer pose on particular instances of objects. Given a particular instance of an object, say an automobile of a certain model and year, an instance-level network will infer the pose of this object. Applying this network that was trained on a particular instance of an object to similar, but not identical instance, is not instance-level pose estimation. Typically, when performing experiments, the exact objects used during training are the objects that are used for testing, but in new poses that were not present at train time. Some instance-level datasets include YCB [4] and LineMod [5]. Some example objects from the YCB dataset include a mustard bottle, crackerbox, bleach bottle, etc. The exact same mustard bottle used in the training set is the same bottle used in the testing set. Trying to use another brand of mustard or using a
bottle with different shape or size to test a network trained on the bottle used in the training set is invalid within the scope of instance-level pose estimation, as this would break the assumption that the object used during training is identical to the object used in testing.

Figure 2.1: The different objects within the food category of the YCB dataset [4].

Figure 2.2: The different objects within the tools category of the YCB dataset [4].

Deep neural networks have revolutionized computer vision. Applying the gradient descent algorithm [6] to optimize parameters iteratively over large datasets has led to massive improvements over traditional computer vision techniques that use hand-crafted features [7]. Problems such as image classification, object detection, and semantic segmentation, among others have been dominated by deep convolutional neural networks [7][8][9], and pose estimation is no exception [2][1][10][11][12]. Typically, these neural networks process examples of the task in the form of input and output pairs. The network makes a prediction and this prediction is compared to the ground truth to quantify how wrong the
prediction was. A loss function is a mathematical equation that quantifies how wrong
the prediction was. The various weights and biases of the neural network control how it
responds to input data, and the loss function is minimized with respect to the network pa-
rameters using gradient descent [6]. This paradigm of learning tasks from labelled data is
called supervised learning.

In deep object pose estimation, supervised learning is common. A network processes
an image of an object, and the network makes a prediction of the object’s pose. This pose
can take the form of a rotation matrix with translation vector, a quaternion with a translation
vector, bounding box coordinates, among others. Xiang et al. use a convolutional network
to regress pose information directly from images [1]. Trying to regress pose parameters
directly from images is difficult, due to scaling ambiguities lost by image projection. Fur-
thermore, the non-linearity of rotation makes it very difficult to regress for convolutional
neural networks [11]. Peng et al. use a pipelined approach to solve the pose estimation
problem [11]. Instead of directly learning the pose parameters, they instead learn to predict
the image coordinates of 3D keypoints on objects. Prior to training, they define a set of 3D
keypoints on each object, and they use a convolutional neural network to predict the pro-
jected locations of these keypoints in the 2D image [11]. Given the 2D-3D correspondences
of these keypoints, they frame the problem as a perspective-n-points problem. Using the
correspondences and the camera intrinsics, perspective-n-points can solve for the optimal
pose of the object in the camera frame. The work gave a fresh perspective on the problem,
and it addressed issues with object occlusion since they used a voting scheme to find the
2D keypoint correspondences [11].
Figure 2.3: An image of the architecture from PoseCNN [1]. This architecture regresses pose directly from RGB images. Image taken from [1].

Figure 2.4: The pipelined approach of PV-Net [11]. This architecture exploits the effectiveness of 2D convolutions to find 2D-3D correspondences. Given these correspondences, a PnP solver calculates the pose. Image taken from [11].

So far, the previous approaches have only used images to solve the problem. This is
rather limited, as 2D projections of the world onto images can result in the loss of information, especially as it pertains to object pose. Using depth information from relatively low-cost RGB-D sensors has led to some promising results for supervised pose estimation. Qi et al. uses 2D detections to isolate pointclouds of objects before performing pose estimation via bounding box parameter regression using pointnet [13], which is a special neural network architecture that can operate directly on pointclouds; this architecture is detailed in section 2.2.1. Wang et al. combine features from the color feature space with features from the 3D pointcloud of the object obtained from the depth image in Densefusion [2]. The fusion of the color features extracted from the convolutional branch with the pointcloud features extracted from the pointnet branch gives a combined point-wise feature that resolves pose ambiguities for symmetric objects. After computing these point-wise fused features, further convolutional layers regress quaternions and translation vectors [2]. This initial estimate is further refined by an iterative-refinement network similar to the upstream Densefusion network. Densefusion outperformed previous works on the YCB dataset, while simultaneously inspiring future papers with its RGB-depth-fused feature representation. Yet another paper that uses a fusion technique is PVN3D [10]. He et al. uses the same Densefusion technique to setup a pipelined approach similar to [11]; however, this algorithm regresses votes of 3D keypoints instead of 2D keypoints, and uses least squares on 3D-to-3D correspondences instead of using a Perspective-N-Points [14] solver on 2D-to-3D correspondences [10].

2.1.2 Category-Level Pose Estimation

Now that instance-level pose estimation has been explained, category-level pose estimation will be covered. This is an important niche area of pose estimation because it strengthens the applicability of pose estimation. A major limitation of instance-level pose estimation is that these algorithms are best suited to operate on one particular object and may not perform well when used on other objects that are similar but not identical.
Category level pose estimation focuses on achieving pose estimation on a particular category of objects. Instead of attempting to perform pose estimation on a particular instance of an object, this family of methods instead attempts to learn a mapping from a generic representation of an entire category to a pose estimate. Using the automobile example from the previous section, this would entail predicting the pose of the entire category of automobile as opposed to predicting the pose of one particular automobile. There exists far fewer works in this family of deep object pose estimation networks than that of instance-level pose estimation. Some datasets that present object-level poses are NYU V2 [15] and the CAMERA [16] datasets. These datasets provide 3D bounding boxes or 6D pose information on several categories of objects such as bottles and digital cameras.

Figure 2.5: Here are some images of some camera objects from the camera dataset [16]. Instead of having completely unique objects in several unique poses, this dataset contains several unique instances of a particular category.

Category-level approaches must learn to operate on several instances of a particular cat-
category instead of only one instance. Wang et al. create a normalized object coordinate space (NOCS) to encode a category of objects [16]. Given an RGB-D image of an object, this algorithm attempts to project each pixel of the object of interest from the color image to the corresponding x, y, z coordinate in the NOCS representation. This method learns a mapping from category instance to generic instance, and then learns to fuse that mapping with the depth image to predict 6D poses. In a follow-up paper, the authors extend the NOCS framework to predict the pose of articulated objects; that is, the pose of a main object, and its constituent parts that move freely [17]. They abandon the RGB component of the original approach, and instead operate entirely on the object pointcloud using pointnet++ [17]. This approach gives the pose of the base object as well as the pose of the articulated sub-objects attached to the base object.

Figure 2.6: Architecture for the normalized object coordinate space network [16]. The 2D convolutional layers in this network map rgb pixels to positions within the NOCS coordinate system. The depth image is then used to project the NOCS representation to real-world positions. Image taken from [16].
Figure 2.7: This figure shows the follow up work of NOCS [17]. Here, the authors discard the rgb information completely and rely on 3D processing to extract pose and articulation information from the scene. Image taken from [17].

2.2 Point Cloud Manipulation

The previous two sections gave an overview of some key methods within deep 6D object pose estimation. Some of the state-of-the-art methods [2] [10] operate on pointclouds to infer the pose of the objects. These methods leverage specialized neural networks that can operate on 3D pointclouds of detected objects. In later chapters in this thesis, a pointcloud-based approach will be introduced in an attempt to improve upon the state-of-the-art for category-level pose estimation. A key principle in that method is the idea of pointcloud manipulation. In this section, some neural network architectures dealing with manipulating
2.2.1 Pointnets

Pointnets are a type of neural network that can directly operate on pointclouds [18]. Instead of having to perform a discretization from a pointcloud into a voxel grid, which would require additional memory to represent an occupancy grid denoting free and occupied cells, pointnet can directly consume sets of points and transform them into a feature representation [18]. A fundamental challenge with processing pointclouds is that they do not assume a regular structure. Unlike a 2D image, a pointcloud does not adhere to any underlying structures, it is simply a set of points. Furthermore, the same pointcloud can be represented by all possible permutations of point orderings in the point set. This breaks any structural assumptions that allow 2D convolutions to operate on images and even 3D convolutions to operate on voxel grids. Pointnet handles these challenges by using pointwise convolutions followed by symmetric non-linear activation functions such as max and average pooling [18]. Regardless of the ordering of points in the pointcloud, the max and average pooling functions do not change the result of these operations. Pointnets are also designed such that they can handle varying numbers of points between multiple pointclouds. These networks generate powerful feature representations that allow for downstream classification, segmentation, regression, adversarial generation, and pose estimation of pointclouds [18].

Figure 2.8: Architecture of pointnet [18]. Image taken from [18].
Point completion network leverages pointnet to perform pointcloud completion [19]. That is, given a partial view of a pointcloud, this neural network can fill in the rest of the points that were initially hidden from view. Yuan et al. [19] encode a latent feature vector containing pointcloud information from the incomplete pointcloud. They then take this latent feature vector and directly regress coordinates of a coarse pointcloud representing the completed pointcloud. This initial pointcloud is coarse, and does not contain finer details representing the object shape. A secondary, post-processing network refines this initial coarse pointcloud and enhances the completion. The authors use this network to perform completions on car pointclouds from the KITTI [20] dataset.
Figure 2.10: Architecture for point completion network [19]. This architecture consists of an autoencoder. The feature extractor for the encoder is a pointnet, while the decoder is a two-stage decoder. Stage one of the decoder is simply a series of fully connected layers that map the latent variable to a coarse-grained reconstruction. Stage two uses grid deformations inspired by folding-based decoders [21] to refine the output of the first stage.

LOGAN [22] performs an unpaired, shape-to-shape translation of one pointcloud to another. This network uses a pointnet and generative-adversarial network [23] (GAN) to learn to translate between shapes of pointclouds. Given an input pointcloud and a target domain, this network can transform the pointcloud to another domain while preserving certain shape features of the original pointcloud. This network is trained in an unpaired fashion, meaning that unlike the loss function used in PCN [19], this network is trained using an adversarial loss that doesn’t explicitly seek to enforce shape similarities between the input and output pointcloud.
Figure 2.11: Here are some example results from the LOGAN [22] network. The point-cloud on the far left show inputs of furniture. The images to the right show the translated pointclouds. They’ve been transformed from chair to tables and tables to chair in a differentiable way.
CHAPTER 3

NOVEL APPROACH TO CATEGORY-LEVEL POSE ESTIMATION

In the past two chapters, the problem of pose estimation was defined and motivated by giving some important example use cases and citing work which approach those problems by using deep neural networks. A brief overview of the field of deep 6D object pose estimation and point cloud manipulation was also given. In this chapter, a key challenge to pose estimation will be explained, and the core idea behind this thesis will be presented. This will be achieved by some discussion about the use cases of these algorithms and how they could be used outside of lab environments.

3.1 Instance-level Pose Estimation in the Wild

As shown in the previous chapter, instance-level pose estimation is quite mature, and shows great results on recent datasets. Current methods [2][10][11] show that when the target object is only a single instance, accurate pose estimation algorithms perform well; however, the setting in which these algorithms could thrive is rather limited. Densefusion [2] gives great performance on the YCB dataset [4]. This dataset consists of single instances of objects. One of the objects in the dataset is a bottle of mustard, and it is the only bottle of mustard that is used. When the model infers pose on this single instance of a mustard bottle, it performs well; but, what would happen when if a similar bottle of mustard with slightly different shape were to appear before the pose estimation algorithm? Would it still perform as well? The following sheds some light on these interesting questions. In Joffe et al [3], the authors demonstrate a robotic system that can infer the pose of chicken carcasses for the purposes of automated handling of chickens. The goal of the work is to have a robot manipulator grab a chicken, and place the chicken into a canonical pose. The chicken appears initially in some random pose within the work space of the robot [4]. The
authors use an augmented-autoencoder [12] to acquire the pose of the chicken, grab the chicken, and orient the chicken in such a way to place it into the canonical pose. They report moderate success with their experiments; however, in some cases, the augmented autoencoder appears to completely fail to give the correct pose [3]. Due to the difficulty involved with labelling all possible perturbations of 6D poses, the authors use a single CAD model of a chicken to train the pose estimation network, and report great results with pose estimation on rendered images; however, when using the network that was trained on the CAD model to infer pose on real chickens, the results were worse than they were on the CAD model images. Now, one could argue that the error in generalization could be attributed to the domain gap between real and synthetic; however, Sundermeyer et al. show that the augmented auto encoder reports acceptable generalization between the synthetic dataset and real dataset [12]. The authors use CAD models of objects from the TLESS dataset [5], a dataset that contains many texture-less objects, to train the networks, and then perform inference on the real-world objects. So, perhaps the domain gap between the real and synthetic settings is not the true cause of this error. The next issue that comes to mind when examining the generalization problem is the variation between the object that the network saw at train time and the object that the network saw at test time. Although instance-level methods perform well on the exact same object that was seen in training, these methods do not seem to transfer well when the object exhibits variation. In the next chapter, this thesis demonstrates an experiment that shows the Densefusion [2] method, which seems to perform accurate pose estimation on the object for which it was trained, but has to generalize to similar objects of the same category. The key idea emphasized here is: instance-level methods do not transfer well to objects of similar category, but with different appearance. This is a significant problem because, in order to perform pose estimation with these instance-level methods, objects at test time need to be near identical to those seen at test time.

In settings where objects of a particular category exhibit significant variation, using
instance-level pose estimation methods seem to be inappropriate. However, in industrial settings, where objects are precisely machined and share mostly homogeneous appearance, instance-level pose estimation may fit perfectly. Category-level pose estimation would be more appropriate for settings in which objects are expected to show significant variation in appearance since these methods directly address the challenge of within-class variation. Even though these methods show promise in determining the pose of categories of objects, they may not perform as well as instance level methods if the number of objects that comprise the category is small. Perhaps by enforcing these networks to learn what shapes constitute an object category, some performance is sacrificed.

### 3.2 Proposed Approach

The proposed approach investigated in this thesis is to reduce the category-level pose estimation problem to a problem of object canonicalization. The approach is as follows: train an instance-level pose estimation method on a canonical instance of an object category, then train a neural network to transform a given input from a given instance to the canonical instance. Next, apply the instance-level method trained on the canonical object to predict the pose of the transformed object, which should appear similar to the canonical instance used in training. This method essentially attempts to offload the generalization expected from instance-level methods to an auxiliary network that focuses solely on canonicalization. This auxiliary network acts as a frontend preprocessing step that makes objects appear as close to the canonical instance as possible. After this preprocessing, the transformed object is sent to the instance-level network, which is known to perform well on single instances of objects. What makes this approach different than other category-level methods is the attempt to leverage already proven instance-level methods. Instead of trying to predict pose from some neural representation of a category, this method attempts to map category pointclouds to canonical pointclouds, which can be directly consumed by instance-level algorithms.
What does the canonicalization step look like? In this thesis, we assume 3D inputs in the form of hundreds of pointcloud triplets. This pointcloud is the medium in which canonicalization operates. This thesis details multiple experiments where pointclouds are consumed by 3D neural networks and transformed into alternate representations. The canonicalization is performed by these 3D neural networks. The networks are given a category instance, and are asked to reconstruct the canonical representation of the object while preserving the pose of the original input. To select which pointcloud would be the canonical representation for the category, experiments were run to see which model the canonicalization methods gave the best results. Figure 3.2 gives an illustration of the canonicalization pipeline. In the next chapter, experiments are presented that detail the experimental design and results of this canonicalization step.

![Diagram of canonicalization pipeline](image)

**Figure 3.1:** Flow of applying instance-level methods to category problems.
Figure 3.2: Flow of the proposed method.
CHAPTER 4
EXPERIMENTAL DESIGN

The objective of this chapter is to explain the design of experiments. First, some experiments are shown that demonstrate the saturation of transferability of one particular instance-level method to similar objects of the same category. Next, a series of experiments are presented that attempt to perform the canonicalization step. Finally, one of the canonicalization methods is paired with an instance-level method to close the loop on the thesis hypothesis, which, again, is using canonicalization to outperform instance-level methods on category-level pose estimation. All experiments detailed here were performed in Ubuntu 16.04 using PyTorch and Tensorflow powered by a GTX GeForce 1070 graphics card. The dataset used was the ShapeNet [24] dataset. This dataset consists of 3D pointclouds of various categories of objects such as chairs, lamps, cars, etc.

4.1 Instance Level Method in Category Setting

Here, a state of the art instance-level method was applied in a category setting. Densefusion will be used here. As stated earlier, this thesis focuses on processing 3D information in the form of pointclouds. Densefusion operates on RGB and depth images, so for this experiment, the network architecture was modified to solely work on depth. To validate this modification, the modified network was trained on some pointclouds of the ShapeNet [24] dataset and tested on held-out data. The color processing branch of Densefusion resolves symmetric shape ambiguities, so asymmetrical objects were chosen to evaluate the depth-only Densefusion network.
4.2 Canonicalization Experiments

This section details a series of canonicalization experiments. Here, the pointcloud manipulation networks are used to attempt canonicalization, and the results are shown. As explained earlier the canonicalization step consists of manipulating the input pointcloud such that it appears as close to the canonical pointcloud as possible. Here, each pointcloud manipulation method is given this task. Figure 4.3 shown below gives an example for the
airplane class of the ShapeNet dataset.

Figure 4.3: This figure shows canonicalization working on some pointclouds of airplanes from the ShapeNet [24] dataset.

4.2.1 Basic Pointnet Autoencoders

For the first method of canonicalization, a simple encoder-decoder neural network was used. The setup for this experiment is similar to 2D convolutional auto encoders for images; here, the input pointcloud takes the place of the input image, and the canonicalized pointcloud is the target image. Two architectures are used here. The first model uses a pointnet encoder to compute a latent-space variable for the input pointcloud. This latent-space variable is computed via the point-wise convolutions of pointnet. After the last convolutional layer, the point-wise features are collapsed along the point dimension via a max-pooling operator as in the original paper [18]. This transforms the output feature shape from (B,
(F, N) to a fixed-size vector of shape (B, F). Here, B is batch-size, F is feature depth, and N is the number of points. The decoder for this network consists of three fully-connected layers that map the latent-space vector to the canonical pointcloud. The second architecture attempts to create a more complete latent-space variable by concatenating the first latent variable with each point-wise feature. After this concatenation, these globally-augmented point-wise features are passed through one additional convolutional layer before a second latent-space variable is created through the same max-collapsing procedure as the first pointnet architecture. For both of these networks, chamfer loss is used to optimize the parameters of the networks. Chamfer loss is represented by equation 2 below:

$$d_{CD}(S_1, S_2) = \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|^2_2 + \sum_{y \in S_2} \min_{x \in S_1} \|x - y\|^2_2$$

(2)

Here, $S_1$ and $S_2$ are two 3D pointsets. $x$ and $y$ are points within the two sets. For each point in one set, the point in the other set that minimizes their L2 norm is chosen. This is done for all $x$ points with respect to $y$ and all $y$ points with respect to $x$. Each norm in the two loops are summed, and then the terms from each loop are summed, giving the chamfer distance.

### 4.2.2 Point Completion Network

This method follows an architecture that is very similar to the prior. In its original setting, this work takes incomplete or partial pointclouds and attempts to fill in the rest of the pointcloud, completing the shape. The network has an encoder-decoder structure similar to the previous autoencoder, but with a more complex decoder. This network uses a two-stage decoding scheme to achieve the completion task. The encoder combines the global pointcloud features with pointwise features from the previous layer. These globally-augmented point-wise features are then passed through one final convolutional layer before
a point-wise max pooling operation computes the final feature vector. An initial decoder uses fully-connected layers to perform a coarse-grain reconstruction of the input point-cloud. Up until this point, this network and the previous method for canonicalization are identical. Where this method diverges is in the addition of another decoder. This decoder is responsible for fine-grain reconstruction of the pointcloud. This stage sits behind the previous decoder and refines the reconstruction from the coarse-grain stage. Whereas the coarse-grain stage used simple fully-connected layers for the reconstruction, the fine-grain stage uses folding-based decoder [21] to achieve a more precise reconstruction. Yuan et al. report that this two-stage method works better than using either method alone. After the pointcloud is reconstructed, a chamfer loss is taken on the input and output pointclouds.

4.2.3 LOGAN: Latent Overcomplete Generative Adversarial Network

This canonicalization method uses two networks to achieve the pointcloud reconstruction task. The original method trains an autoencoder and a generative adversarial network to translate shapes from one domain to another domain. The autoencoder of this method uses pointnet++ to encode multi-scale features of an input pointcloud. The encoder outputs four feature vectors of length 64. Each vector is encoding information about the pointcloud at increasing scales. The four vectors are then concatenated to form an overcomplete latent vector that contains information about the pointcloud. The decoder for the autoencoder consists of four fully-connected layers. The output of these layers calculate the coordinates of each point within the pointcloud. The output pointcloud is then compared with the input pointcloud, and the Earth Mover’s Distance is used as the loss function. This autoencoder is trained until sufficient reconstructions are achieved.

Next, the GAN is trained using the same pointclouds and the encoder from the trained autoencoder from the previous step. After the previously trained encoder gives a latent code of the input pointcloud, the GAN attempts to map the latent representation of a pointcloud from one domain to a pointcloud of a different, but related domain. Once the GAN is
trained, the two networks can produce latent codes of an input pointcloud and then translate then latent codes to another domain.

### 4.2.4 3D Convolutional Autoencoder

For the final canonicalization architecture, 3D convolutions are used. The prior three architectures operate on 3D data represented as pointclouds; however, this experiment preprocesses input pointclouds into 3D voxel grids before feeding the data into the network. Similar to the pointnet autoencoder, multiple architectures were used for this method. The first method uses a basic encoder-decoder network. The encoder is comprised of 3D convolutions, and the decoder uses 3D transpose convolutions to differentiably upsample the features from previous layers back to original voxel size. The second architecture uses skip connections to combine high-level location information from early layers with low-level semantic information from later layers. This is inspired by Unet [25], which is a state-of-the-art network used for image segmentation. Because this architecture uses voxel grids as opposed to pointclouds, binary-cross-entropy loss is used to optimize the network. The ground-truth voxel grid has points in cells that have occupancy and no points in cells that are free. The cells with occupancy have a value of one, whereas the points with no occupancy have a value of zero.
Figure 4.4: The 2D convolutional Unet [25]. The 3D autoencoder is identical, the only difference is the use of 3D convolutions as opposed to 2D. Image taken from [25].
CHAPTER 5
RESULTS AND DISCUSSION

This chapter will present tables and figures showing the results of the experiments described in the previous chapter.

5.1 Densefusion Modification Verification Experiment

<table>
<thead>
<tr>
<th>Category</th>
<th>ADD ≤2cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair</td>
<td>92.5</td>
</tr>
<tr>
<td>Airplane</td>
<td>91.7</td>
</tr>
<tr>
<td>Table</td>
<td>93.4</td>
</tr>
<tr>
<td>Car</td>
<td>93.5</td>
</tr>
</tbody>
</table>

Table 5.1: Densefusion Modification Validation

This experiment verified the modification of Densefusion. To reiterate, Densefusion uses both 2D convolutions and pointnet to extract and fuse together color and 3D features [2]; however, the setup of the canonicalization step only uses 3D inputs. To allow this to work, the color branch was deleted, and the 3D branch was kept. This modification is shown in figure 4.2. Before attempting to use Densefusion to perform inference solely on 3D pointclouds in the category-level setting, an experiment was conducted to see how this modified Densefusion network would behave in its native instance-level setting. Table 5.1 shows the results of the modified, depth-only Densefusion network performing inference on single instances of four different categories from the ShapeNet dataset [24]. In table 5.1, the right column shows the quantitative results using the average model point distance metric (ADD), which was used in the original Densefusion paper [2]. This metric is represented by equation 3 below.
Here, $R$ and $\hat{R}$ are the groundtruth rotation and predicted rotation, respectively. $T$, and $\hat{T}$ are the groundtruth and predicted translation respectively. The $x$ is the set of 3D points from an object model. Both point sets are transformed using these rotations and translations, and then the norms of the two point sets are taken and averaged. In table 5.1, the right column is showing the percentage of evaluation inferences whose ADD metric is $\leq 2cm$. The scores in this table are comparable to scores found in the original Densefusion results [2], so the modification is considered valid here. Additionally, these scores serve as an upperbound for what further category-level experimental results can produce.

### 5.2 Canonicalization

<table>
<thead>
<tr>
<th>Object Category</th>
<th>PN AE A</th>
<th>PN AE B</th>
<th>PCN</th>
<th>LOGAN</th>
<th>3D Conv AE A</th>
<th>3D Conv AE B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair</td>
<td>0.288</td>
<td>0.270</td>
<td>0.282</td>
<td>0.590</td>
<td>0.190</td>
<td><strong>0.183</strong></td>
</tr>
<tr>
<td>Airplane</td>
<td>0.307</td>
<td>0.292</td>
<td><strong>0.276</strong></td>
<td>0.588</td>
<td>0.280</td>
<td>0.283</td>
</tr>
<tr>
<td>Table</td>
<td>0.331</td>
<td>0.318</td>
<td>0.297</td>
<td>0.498</td>
<td>0.293</td>
<td><strong>0.289</strong></td>
</tr>
<tr>
<td>Car</td>
<td>0.256</td>
<td>0.228</td>
<td><strong>0.205</strong></td>
<td>0.416</td>
<td>0.220</td>
<td>0.216</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Category</th>
<th>PN AE A</th>
<th>PN AE B</th>
<th>PCN</th>
<th>LOGAN</th>
<th>3D Conv AE A</th>
<th>3D Conv AE B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair</td>
<td>0.0289</td>
<td>0.0236</td>
<td>0.0207</td>
<td>0.0590</td>
<td>0.0197</td>
<td><strong>0.0189</strong></td>
</tr>
<tr>
<td>Airplane</td>
<td>0.0204</td>
<td>0.0217</td>
<td>0.0191</td>
<td>0.0469</td>
<td><strong>0.0173</strong></td>
<td>0.0195</td>
</tr>
<tr>
<td>Table</td>
<td>0.0382</td>
<td>0.0305</td>
<td>0.0282</td>
<td>0.0488</td>
<td>0.0199</td>
<td><strong>0.0189</strong></td>
</tr>
<tr>
<td>Car</td>
<td>0.0266</td>
<td>0.0228</td>
<td><strong>0.0198</strong></td>
<td>0.0326</td>
<td>0.0200</td>
<td>0.0199</td>
</tr>
</tbody>
</table>

In these experiments, a single instance of a category was chosen as the canonical model, and then the other 500 object models were used to train the canonicalization methods. The
canonicalization methods performed poorly except for a limited range of angles as shown in tables 5.2 and 5.3. When uniformly sampling angles from the entirety of SO(3), which is the set of all rotations with respect to the origin of 3D Euclidean space, no methods seemed to be able to learn the task of pointcloud canonicalization. All of the methods output pointclouds that had vague shapes of the target object, or failed entirely, giving a spurious collection of points. An example of this is shown in figure 5.3. When limiting the angle of rotation, the networks perform much better. Limiting the magnitude of the rotation in the range of 5-10 degrees in any axis gave much better results. Figures 5.1 and 5.2 show ideal canonicalizations that resulted from limiting the rotation range.

The simple autoencoder gave decent results in the canonicalization task. Despite its simple architecture, the pointnet encoder and fully-connected decoder worked well to canonicalize pointclouds with minor rotations. The pointnet autoencoder B architecture seemed to outperform the A architecture. The concatenation of the level 1 global feature with the pointwise features may have given more context for the decoder.

The point completion network was more or less identical to the pointnet autoencoder B architecture. The only difference was in the decoder. The decoder in this method employed a two-stage decoding scheme. The fine-grained decoder used by the point completion network likely gave the boost in performance over the single-staged decoding process of the pointnet autoencoder.

The LOGAN had a much more involved architecture, employing both an autoencoder and a GAN, but did not fare well compared to the other methods. The big difference between the autoencoder in this work was that it used pointnet++ to extract 3D features from the input pointcloud and created a stacked latent-space variable that attempted to capture 3D information at multiple scales. Another major difference that separates this network from the others is the method for pointcloud manipulation. Instead of setting up an encoder-decoder network to change cloud A to cloud B, this method uses a GAN to perform the translations. Perhaps this task with the varying objects under different rotation
was too much for the GAN to capture in its translation, and a more direct translation is better suited.

The best performing network was the 3D convolutional network. This network likely performed better than the others due to the fundamental difference in its structure; it was the only algorithm to not use pointclouds to represent the 3D data. When using the voxel grid representation, the network contains connections to all possible output voxels due to the strided, 3D convolutions. Conversely, the models that utilize the pointnet must regress the coordinates of the 3D points of the output pointcloud and form the structure of the target object, which may be much a more difficult task since there is not a predefined set of locations to infer. The inclusion of the skip connections in the 3D convolutional autoencoder improved the canonicalization performance as well. This is likely due to the fact that the upsampling convolutions can glean useful information from the features of earlier layers. Unlike the pointnet-based models, the 3D convolutional autoencoder performed the decoding in a convolutional way. This allowed the decoder to capture 3D structure in the voxel grid similar to how 2D image patterns are detected by 2D convolutions. The decoders of the previous models employed simple fully-connected layers to map the latent code representation of the input pointcloud to the reconstructed output, which probably makes capturing structure quite difficult for that part of the network.
Figure 5.1: Reconstructions of cars from the ShapeNet dataset [24]. The 3D Convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds are rotated with rotations taken from the limited SO(3) distribution.
Figure 5.2: Reconstructions of chairs from the ShapeNet dataset [24]. The 3D Convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds are rotated with rotations taken from the limited SO(3) distribution.
Figure 5.3: Attempted reconstructions of cars and chairs from the ShapeNet dataset [24]. The 3D convolutional autoencoder with skip connections was used for these reconstructions. Here, the pointclouds were rotated with rotations taken from the uniform SO(3) distribution. Here, the 3D convolutional autoencoder fails to reconstruct the pointcloud.

5.3 Category-Level Pose Estimation via Canonicalization

Tables 5.4 and 5.5 below show head-to-head comparisons of using Densefusion with and without canonicalization. In both tables, the without column shows the result of the modified Densefusion network performing inference in the category-level setting. The purpose of this experiment was to test the capacity of this instance-level architecture. Reiterating earlier points, the original setting in which this model is used is instance level. In this setting, the model is given x objects from a particular category to train on. After this training is complete, the model must perform inference on the held out models out of the original 500 models. For example, the row Airplane-50 in table 5.4 shows the results of Densefusion trained on 50 different airplane models in various poses in SO(3), and the corresponding
cells in the other columns show the metric on the models inferring pose on the remaining 450 airplane models, which are different but come from the same distribution of poses from SO(3). The second and third columns show the percentage of predictions on the number of models whose ADD was $\leq 2\text{cm}$.

Table 5.4: Densefusion With and Without Canonicalization - Uniform SO(3) - ADD $\leq 2\text{cm}$

<table>
<thead>
<tr>
<th>Category</th>
<th>Without Canon</th>
<th>With Canon (3D Conv AE B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair - 10</td>
<td>72.2</td>
<td>3.2</td>
</tr>
<tr>
<td>Chair - 50</td>
<td>80.7</td>
<td>2.2</td>
</tr>
<tr>
<td>Chair - 100</td>
<td>83.3</td>
<td>1.8</td>
</tr>
<tr>
<td>Airplane - 10</td>
<td>80.2</td>
<td>4.7</td>
</tr>
<tr>
<td>Airplane - 50</td>
<td>86.0</td>
<td>3.2</td>
</tr>
<tr>
<td>Airplane - 100</td>
<td>88.3</td>
<td>0</td>
</tr>
<tr>
<td>Table - 10</td>
<td>70.7</td>
<td>5.5</td>
</tr>
<tr>
<td>Table - 50</td>
<td>77.1</td>
<td>2.0</td>
</tr>
<tr>
<td>Table - 100</td>
<td>80.3</td>
<td>2.3</td>
</tr>
<tr>
<td>Car - 10</td>
<td>78.7</td>
<td>4.8</td>
</tr>
<tr>
<td>Car - 50</td>
<td>82.4</td>
<td>3.6</td>
</tr>
<tr>
<td>Car - 100</td>
<td>83.9</td>
<td>2.7</td>
</tr>
</tbody>
</table>

Table 5.4 shows the effect of canonicalization in the setting of unrestricted rotational displacement of the pointclouds. Because the canonicalization methods performed so poorly in the unrestricted experiments, the pose estimation for this setting was doomed. The output pointclouds from the models did not resemble the target canonical shape at all most of the time. Since the canonicalization preprocessing was nonsensical, the input pointclouds fed into Densefusion caused very poor performance. Figure 5.3 shows an example of what a poorly canonicalized model can look like before being fed into the modified Densefusion.

Table 5.5 shows the same experiment as the previous one; however, now the rotation that the pointcloud undergoes is limited. After seeing the poor performance of canonicalization of completely randomly rotated pointclouds, another experiment was conducted using a restricted range of rotation, just like in the canonicalization experiment. The Without Canon column of table 5.5 showing limited generalization to novel category instances.
Table 5.5: Densefusion With and Without Canonicalization - Limited SO(3) - ADD $\leq 2$cm

<table>
<thead>
<tr>
<th>Category</th>
<th>Without Canon</th>
<th>With Canon (3D Conv AE B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair - 10</td>
<td>73.8</td>
<td>71.7</td>
</tr>
<tr>
<td>Chair - 50</td>
<td>80.2</td>
<td>82.1</td>
</tr>
<tr>
<td>Chair - 100</td>
<td>82.7</td>
<td>84.6</td>
</tr>
<tr>
<td>Airplane - 10</td>
<td>83.2</td>
<td>78.4</td>
</tr>
<tr>
<td>Airplane - 50</td>
<td>85.7</td>
<td>86.8</td>
</tr>
<tr>
<td>Airplane - 100</td>
<td>88.2</td>
<td>87.3</td>
</tr>
<tr>
<td>Table - 10</td>
<td>71.6</td>
<td>72.0</td>
</tr>
<tr>
<td>Table - 50</td>
<td>79.0</td>
<td>78.8</td>
</tr>
<tr>
<td>Table - 100</td>
<td>81.7</td>
<td>82.3</td>
</tr>
<tr>
<td>Car - 10</td>
<td>79.3</td>
<td>80.5</td>
</tr>
<tr>
<td>Car - 50</td>
<td>83.4</td>
<td>84.2</td>
</tr>
<tr>
<td>Car - 100</td>
<td>86.9</td>
<td>86.1</td>
</tr>
</tbody>
</table>

exposes a possible saturation of this model. However, when first performing the canonicalization with the 3D convolutional autoencoders or the point completion networks, this saturation problem is reduced. This point is supported when looking at averages across the number of categories when the training set size is 50 and 100 for the model predicting with canonicalization. Across all categories, the ADD $\leq 2$cm averages for categories with train size 50 and 100 with canonicalization are 82.975 and 85.075, respectively. This is slightly higher than the averages without canonicalization, which are 82.075 for 50 and 84.875 for 100. Because the responsibility of computing the various instances of categories is dedicated to the canonicalization networks, the instance-level pose estimation algorithm can focus on predicting the pose of the canonical object of the category. Comparing the two columns in table 5.5, one can see slight improvement when using the canonicalization frontend paired with the modified Densefusion. This improvement is very slim, and the setting in which it is applied is very constrained, however.
CHAPTER 6
CONCLUSION AND FUTURE WORK

This thesis has details the pose estimation problem and discussed some key challenges within the field. An overview of state-of-the-art methods for performing pose estimation was given, along with the introduction of the niche subfield of category-level pose estimation. A novel approach to category-level pose estimation was demonstrated with limited success along with a brief discussion describing how key differences in the various techniques affected the experimental outcomes.

Future work for this research could swap out Densefusion for other instance-level pose estimation methods. In this work, significant time was spent modifying the original architecture of Densefusion to work solely on 3D inputs as opposed to both color and 3D information. Additionally, future works could try more complex deep learning techniques for canonicalization such as graph neural networks and neural rendering. An immense roadblock in the applicability for this technique was the fact that it worked in such a constrained setting. Given the superior performance of the 3D convolutional network for canonicalization, one may consider using that type of architecture for performing direct pose estimation in the category-level setting since the feature extractors seemed to have made a meaningful representation. In the experiments, the instance-level pose estimator and canonicalization are setup independently in regards to the canonical model that the two neural networks would use. It was assumed that training the canonicalization to recreate the canonical model and then training Densefusion on that canonical model was valid. Future work could instead first train canonicalization, and then use whatever output from that neural network to train the instance-level network. This could potentially make the instance-level network more robust to the distribution of outputs from canonicalization. Furthermore, future work could setup this pipeline to be trained jointly in an end-to-end fashion. Here,
both networks are trained in isolation, and it is assume that chamfer loss is appropriate for
the canonicalization task. If, however, the output of the canonicalization is fed directly into
the instance-level method, perhaps the gradient descent algorithm could find better ways
to create the canonicalization network. Lastly, future work should take this method and
directly compare against other category-level methods. In this work, the key focus was
on the conversion of instance-level method to category-level via a preprocessing step, and
the nature of the comparisons were only focused on comparing the difference between the
instance-level as-is vs instance-level with the preprocessing frontend. A direct comparison
between other category level methods would be helpful.
REFERENCES


