A LATENT SPACE ANALYSIS OF THE SKETCH-TO-MODEL PROBLEM

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

John Lyons Wolford

December 2022

ABSTRACT

Despite the accomplishments made in the field of computer science, computers still struggle with many tasks that are relatively simple for humans. One such task is to interpret the approximate three-dimensional structure of an object from a ‘sketch,’ which here is defined as a grayscale two-dimensional line drawing. Algorithmic approaches where the rules for interpretation are explicitly defined by the programmer have seen some success, but these methods have a limited ability to account for the many nuances of human perception that influence our interpretation of 3D structure. More recent approaches for solving the related problem of interpreting three-dimensional structure from two-dimensional digital photographs rely on deep learning techniques and large datasets. These approaches have remarkable performance, but many are plagued by artifacts and limitations inherent to the format of the output representation. Research on generating the three-dimensional structure with implicit representations have promise in that they avoid the limitations of the other formats, and the goal of this thesis is to generate these implicit representations from sketches instead of digital photographs.

This thesis proposes a two-step process for constructing three-dimensional geometry from sketches. First, the sketch is encoded into a sketch-model latent space, and second, the sketch’s latent space representation is used to generate an approximate signed distance field of the object. The first step was accomplished by training a convolutional variational autoencoder on a set of synthetically generated line-drawings, and the number of dimensions chosen for the latent space was 1024. Principal Component Analysis was used to get a rough estimate of the necessary number of latent dimensions, and the appropriate dimensionality was confirmed with measurements of

variational autoencoder's test set performance using varying numbers of latent dimensions. The second step was accomplished with an eight-layer network modeled after the architecture presented in [Park19], with modifications to account for the larger dimensionality of the input latent codes. To render the object, this second network was queried repeatedly using a modified sphere tracing algorithm to identify the surface, and an arbitrary directional light source and simple Lambertian shading were used to color the surface.

While the networks performed well considering the naïve approach to generating the latent space and the vast increase in the latent space dimensions compared to [Park19], the quality was insufficient for accurately reconstructing sketch geometries beyond a rough approximation of their convex hull. The low-quality results can be explained by one or more of the following limitations with the approach used in this thesis: a poorly behaved sketch-model latent space, an insufficient amount of training data, suboptimal design choices for the network architectures, and suboptimal methods for training. Possible ways to address these limitations are presented as potential future work.

BIOGRAPHICAL SKETCH

John Wolford was born in Erie, Pennsylvania on May 27, 1998, to Matthew and Marian Wolford, and he spent his childhood years growing up with his three siblings, Claire, Benjamin, and Daniel Wolford. He attended Erie Day School for first through eighth grade, where he first developed an interest in computers, mathematics, science, Shakespeare, and just about everything else his teachers taught him. His interest in programming, however, started during high school when he attended an iD Tech summer camp that taught the basics of C++, Java, and Python over a two-week span. It was after this fast-paced introduction to programming that he began writing code for personal projects, and his interest in computer science turned into a true hobby. By the time he graduated from Northwest Pennsylvania Collegiate Academy, he had taken nearly every computer science course the school had to offer.

Following his graduation from high school, John attended the College of Engineering at Cornell University where he first met Professor Greenberg as a freshman student in his course “Visual Imaging in the Electronic Age.” Professor Greenberg’s opening lecture was the first Cornell lecture John had the pleasure of attending, and he was nearly twenty minutes late because he was waiting the wrong classroom for class to begin. Despite this rocky start to the course, John gravitated towards the subject matter, and he joined the undergraduate research team of the Program of Computer Graphics (PCG) the following semester to continue working with Professor Greenberg. Four years later, in May of 2020, John received a Bachelor of Science in chemical engineering with a minor in computer science amidst the chaos of the global COVID-19 health crisis, and he chose to continue his studies at the Program of Computer Graphics as a master’s student, where this work was completed.

This thesis is dedicated to my family, including my honorary uncle.

ACKNOWLEDGEMENTS

I first thank my close friend, Leul Tesfaye, both for his invaluable assistance in my work for this thesis and for the wonderful friendship we share outside of academics that keeps me motivated to continue pursuing my research interests. Without him, I doubt I would have completed half of what I've now been able to accomplish with this project.

Second, I thank Professor Greenberg for everything he's done for me. From inspiring me to study computer graphics, to willingly sharing his expertise in the interdisciplinary connections between architecture and computer science, to generously helping me craft a path for my future, he has been supporting me for every step of my post-high school academic journey, and for that I am truly grateful.

Professor Bart Selman's advanced artificial intelligence course offered me the opportunity to engage with high level, difficult topics and to think about the limitations of modern machine learning approaches with respect to these topics. As my minor advisor, he provided valuable insight and feedback for my work on latent spaces, and I thank him for his time and involvement in this project.

The Program of Computer Graphics has a history of collaboration across its many members, and I owe a great deal of thanks to the faculty and students who helped me push this project forwards. I'd specifically like to thank Professor Steve Marschner and Dr. Bruce Walter for answering all the many questions I regularly ask them, Jack Otto for his insightful conversations with me about the conceptual nature of the sketch-to-model problem, Sean Brynjólfsson for his unyielding curiosity about my work and his recommendations for speeding up my model generation script, and Isha Chirimar for directing me and other PCG students to some incredible research on the sketch-to-model problem that is now included in my literature survey in Chapter 2 of this thesis.

I would like to thank Professor Bharath Hariharan for teaching two excellent computer vision courses, one of which introduced me to the papers that formed the basis of this thesis. Moreover, I would also like to thank him for the advice he gave me after one of his lectures when I approached him with my idea for this thesis; his recommendation of a convolutional pre-processing step led me to the remarkable research on latent spaces that became the basis for Chapter 4 of this thesis.

Finally, to all those who loved and supported me throughout my studies, your kindness means the world to me, and I thank you from the bottom of my heart.

Table of Contents

Chapter 1

Introduction

The computer, since its earliest days, has been used to quickly complete tasks that were either monotonous or infeasible for humans. At first, this was limited to simple arithmetic; machine computers replaced their human counterparts who would work through the arithmetic calculations by hand. As computational power grew, the computer was used for problems which, though possible to perform by hand, would require an enormous amount of time and effort, and eventually, for problems which were infeasible for a human to complete by hand in the span of his/her life. In the last seventy years, computational power has experienced over a trillion-X speedup, and our focus has shifted to leveraging this fast computation to improve all other disciplines.

Design is a discipline that has arguably been both greatly enhanced by and somewhat left behind by the aforementioned advances in computation. On the one hand, the programs available to designers today enable rapid creative design within the framework of the programs' capabilities. Computer-Aided Design programs like Autodesk 3dsMax or McNeel Rhinoceros3D are simple enough to use that many modern architects and architecture students find it easier to start their designs using these programs rather than on pen and paper. Furthermore, by finding unique ways to apply the computational simulation tools of the modern age, designers have created incredible structures, such as “The MaoHaus” (see [Zheng17]) which would be orders of magnitude more difficult without these simulation tools. Finally, advances in procedural modeling have allowed for human-supervised generative design, where programs create designs based on parametric constraints that are controlled by human users. These designs can be used directly, or they can be modified by the user as desired. In summary, the field of design has undeniably been enhanced by the opportunity to creatively play with the software provided by the increased computational capabilities of modern computers.

On the other hand, the computer still lacks the ability to *interpret* designs. Figure 1.1 shows a series of line drawings that represent 3D shapes. While humans have no issues interpreting the 3D structures represented by these drawings, computers have great difficulty doing so. As far as we at the Program of Computer Graphics are aware, as of the time of writing this thesis, there is no program that can solve the general problem of converting these kinds of line-drawings, drawn from a monocular single point of view, into the 3D objects they represent. Thus, as it stands, if a designer chooses to start their design process with sketching, it is up to him/her to recreate the 3D structure of the design using the CAD tools available. Many designers find this process to be exceedingly tedious and would happily delegate the task to the computer if it were able to automatically generate the three-dimensional geometry from their drawings.

While the definition of a “sketch” is somewhat ambiguous, this thesis will use the term to refer to the types of line drawings shown in Figure 1.1. Sketches that convey 3D geometry are almost always projections of the geometry onto a 2D “picture plane,” and there are a number of possibilities for the type of projection chosen. In this work, the sketches generated for training use perspective projections, though the perspective distortion is often hard to notice due to a narrow field of view. Moreover, sketches may or may not contain hidden lines - lines which are occluded from the camera by the geometry of the 3D object. In this thesis, hidden lines are not included in the sketches used for training, as one of the goals is to gauge the ability of the machine to reconstruct the object without the information provided by these hidden lines. Finally, because the sketches used for this thesis do not include any context, they can only convey a geometric shape; they cannot convey the "scale" of the object they represent.

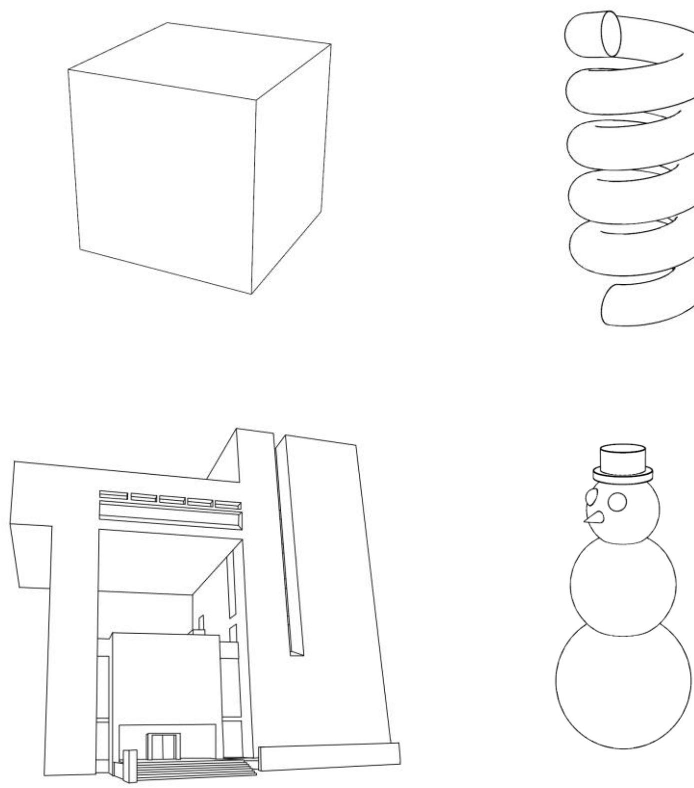


Figure 1.1 Line drawings representing various 3D forms. Although humans can infer the 3D shapes these line drawings represent, the task is extremely difficult for a computer.

The process of interpreting 3D structure from 2D representations is not a new problem. There is a large body of work in the field of computer vision devoted to tackling this problem in the context of digital photographs (see Chapter 2 for more details). Until recently, many of these approaches created artifacts based on the representation format chosen as the output, or they often required multiple viewpoints for reasonable quality results. However, recently published work on generating implicit representations of surfaces has impressed both myself and those advising me, and I believe there is promise in using these implicit representations to solve the sketch-to-model problem.

One of the most important aspects of the approach presented in [Park19], the 3D reconstruction research paper which forms the basis of this thesis, is the use of a *latent space*. In general, a latent space is a space of lower-dimensional encodings for a set of items where similar items have similar encodings. In the context of the 3D reconstruction approach presented in [Park19], the latent space would need to encode all the possible 3D objects that the network can reconstruct. It is therefore critical that this latent space be well-behaved for the task of reconstruction, and developing such a latent space poses a significant challenge.

This thesis attempts to take the first step in bridging the gap between the sketch interpretation problem and modern computer vision techniques for single-view 3D object reconstruction. Over the remaining chapters, I will outline a naïve approach for generating a three-dimensional structure from a two-dimensional sketch. Specifically, I will include discussions about characterizing the dimensions of a sketch-model latent space, developing an approximation of a sketch-model latent space with deep learning techniques, using a sketch’s encoding in the latent space to construct a signed distance field of the object depicted in the sketch, and rendering the surface of the object represented by the signed distance field.

Chapter 2

Related Work

In the previous chapter, I stated that I knew of no programs capable of solving the general case of the sketch-to-model problem. This is not for a lack of trying; given the ubiquitous nature of sketching as a means of conveying ideas, there exists a body of research dedicated to solving the problem of converting sketches to the 3D objects they represent. This research can be split into two separate categories: 1) algorithmic approaches, which rely on pre-meditated, hard-coded rules for generating 3D objects from their sketches, and 2) machine learning approaches, which avoid the need for the programmer to define these rules but rely on large amounts of data. In this chapter, I will outline not only the research which is directly related to object reconstruction from sketches, but also the photographic reconstruction methods and latent space investigations that formed the basis for this thesis.

2.1 Algorithmic Rule-Based Approaches

The earliest research on creating 3D objects from their 2D sketches was done before the wide-scale use of machine learning, and so it is purely algorithmic. While it is impossible to objectively identify exactly who first attempted to algorithmically reconstruct objects from sketches, one of the most rigorous attempts at doing so was that of Dr. Hod Lipson and his collaborators. [Masry05] represents the culmination of roughly a decade of work on the problem, and the paper outlines an interface that, for certain types of sketches, can interactively create rough 3D models based on 2D line drawings. The quality of the results are, in my subjective opinion, quite compelling, and their interface even allows the user to run Finite Element Analysis (FEA) on the resulting models, enabling designers to quickly adapt their sketches to ensure their designs are structurally sound. The papers that formed the basis of the interface include [Lipson95], [Shpitlani95], [Lipson96], [Shpitlani97], [Lipson98], [Lipson00], [Lipson02], and [Kang04]. These papers all provide new insights and approaches to the sketch reconstruction problem which culminate in the interface shown in [Masry05].

The general approach to reconstruction developed by these works is the following. First, a line drawing is transformed into a “connectivity graph” with vertices for each of the stroke endpoints and edges for each stroke. Endpoints which satisfy the overlap conditions described in [Shpitlani97] are combined into a single vertex. Second, the depths of these vertices from the sketching plane are determined by techniques presented in [Kang04] and in [Lipson96]. Third, the strokes between the vertices are reconstructed by assuming that the strokes are contained with a plane (though not necessarily the sketching plane) and solving an optimization problem based on this planar assumption. Each stroke is thought of as a collection of stroke points, and the objective of the optimization problem is to minimize the sum of the squared distances of each stroke point to both of the stroke endpoints. Finally, the faces of the object are created from planar closed loops formed by the strokes, and the final object is rendered.

Although this interface performs exceedingly well for many types of sketches, it does suffer from some underlying requirements that are not always true for all user sketches. One of the worst offenders is that the approach requires all hidden edges be drawn in the sketch, a requirement that is undesirable for designers and often unnecessary for accurate human interpretation. To illustrate this, consider the two images in Figure 2.1. While human observers can interpret these images correctly (albeit with some ambiguity as to the orientation of the leftmost image), Masry's and Lipson's reconstruction process would only interpret the leftmost image correctly. The process relies on the hidden lines of the image to create a graph and subsequent spanning tree, and these lines are not included in the rightmost image. Thus, it would most likely create three attached squares with no backside. This isn't technically incorrect; the projection of the surface formed by the attached squares would also result in the original sketch, but humans tend to interpret the image on the right as a cube because of our visual system's assumptions about the world.

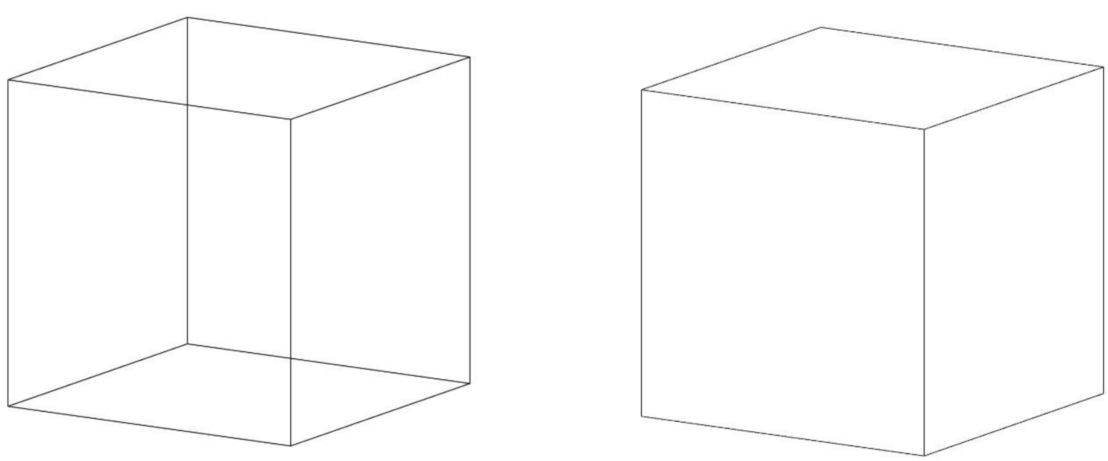


Figure 2.1 An example shortcoming of Lipson's approach. Only the leftmost image would be interpreted "correctly" by Masry's and Lipson's interface as a cube, as it includes the hidden lines necessary for reconstructions by their approach. Note that these sketches are parallel projections, whereas the sketches used for this thesis work are perspective projections.

Another way Masry's and Lipson's approach can fail is in the generation of the aforementioned connectivity graph. When a sketch is drawn from a view where some of the stroke endpoints are aligned, the connectivity graph is impossible to orient correctly, as it collapses all overlapping endpoints into a single vertex in the graph. Unfortunately, while one might hope that a sufficiently complex rule-based algorithm could resolve this problem, determining whether or not endpoints should be combined is a problem that can depend entirely on the semantic meaning of the sketch, as illustrated in Figure 2.2.

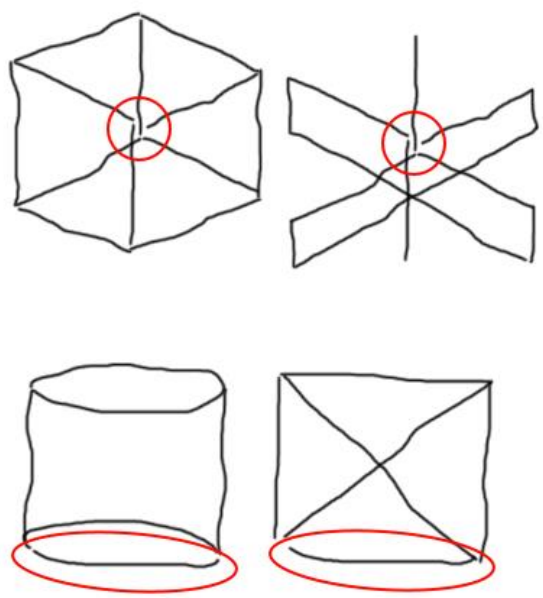


Figure 2.2 Problematic edge cases recreated from Fig. 4 of [Shpitlani97]. In both examples, the correct answer of a local decision depends on an accurate understanding of the entire sketch. In the top example, identical stroke lines meet at a point, and whether or not they should be joined depends on the meaning of the sketch as a whole - the endpoints should NOT be combined for the cube on the left, and the endpoints SHOULD be combined for the propellor on the right. In the bottom example, whether or not the bottom line should be treated as straight or curved depends on an understanding that the left sketch depicts a cylinder and the right sketch depicts a square.

[Cao2005] developed a method for using symmetrical topological principles to generate hidden vertices/edges and to construct perceptually optimal 3D representations for the hidden vertices. The work relied on topological rules to construct candidate sets of hidden vertices, and used the principle of symmetry to 1) determine which candidate set was "perceptually optimal" and 1) orient these hidden vertices. The sketches could then be turned into 3D models using optimization techniques similar in nature to those of Lipson's work. This work does give incredibly promising results, and while it assumes that the surfaces are polyhedral, it doesn't seem that far a stretch to imagine that similar rules could be applied for non-polyhedral objects as well. Unfortunately, this technique also suffers from the same stroke endpoint alignment issues as Masry's and Lipson's work, as it relies on an accurate connectivity graph of the visible strokes, and it could also suffer from over-reliance on the rule of symmetry.

To generalize the last few paragraphs, a major issue with attempting to solve this problem algorithmically is that the algorithm's explicit rules must correctly handle all the assumptions made about sketches by the human visual system. While the work done in the past did an admirable job of addressing and incorporating many of these assumptions into the reconstruction processes, it still could not account for all of them in one, coherent program. This problem is not unique to 3D object reconstruction; many other computer science problems have similarly complicated assumptions that are difficult to enumerate and incorporate algorithmically (Natural Language Processing, image/object recognition, and semantic segmentation to name a few). With the development and wide-spread use of machine learning (and more specifically deep learning) techniques, these nuances can now be "learned" by an appropriate learning algorithm with sufficient data. In the next few sections, we will look at the work that has been done in applying machine learning algorithms to the sketch-to-model problem and related problems.

One last approach I'd like to include before discussing the learning-based related work is that of the BendSketch system presented in [Li17]. This approach does not aim to generate the 3D object representation of an arbitrary sketch, rather, it instead imposes additional requirements for the sketch on the user that allow for reconstruction of arbitrary objects. The user is required to draw and label specific types of stroke lines: boundary, bending, and sharp feature, to represent the object on the sketch plane, and from these stroke lines the object can be generated. While the system doesn't allow for complete flexibility by the user, a trained sketcher can quickly create precise 3D models of their ideas with minimal effort.

2.2 Machine Learning Approaches

Machine learning is now, at least in some measure, at the forefront of nearly every field of computer science, and so it should come as no surprise that the latest research on generating models from sketches relies on machine learning algorithms. As stated in the last section, one of the major issues plaguing early research was that the human visual system's assumptions about sketches were difficult to specify explicitly in an algorithm. Given a large dataset with high-quality data, an appropriate neural network architecture, and ample training time, machine learning techniques can solve problems without having to program any of these difficult to enumerate assumptions. In this section, we review papers applying machine learning approaches to the "realistic image-to-model problem," papers investigating the latent spaces of both sketches and 3D objects, and finally papers applying machine learning approaches to the sketch-to-model problem.

Image-to-Model Approaches:

While the sketch-to-model problem has been investigated by several papers, a much larger body of research has worked on the closely related image-to-model problem, where digital photographs or renderings are used as input in place of sketches. It is this body of work, or specifically the subset of papers introduced to me during my graduate computer vision course, that formed the basis of the work presented in this thesis. The papers presented below all take a supervised learning approach, where one or many images are paired with the associated 3D model to create a training dataset for a neural network. The neural network is then trained on the dataset to predict the 3D model from the input image(s).

One of the challenges of working with three-dimensional objects as the output of a machine learning problem is selecting the appropriate representation format. Computers can represent objects in many different ways, and the representation format that is chosen typically depends on the use case. Representations of 3D objects commonly used in computer graphics include meshes, point clouds, voxel grids, and spline surfaces. While all of these representations work well for certain problems, it is difficult (though by no means impossible) to use them in a machine learning setting. Below I discuss various papers that attempt to reconstruct 3D geometries in each of these formats from 2D images.

- *Meshes*: Meshes are easily the most common format for representing 3D objects in computer science. They are composed of vertices, edges, and faces, and they essentially store the information necessary to describe the geometric surface of the object. They can also include information about the object's surface normals and texture mapping parameters, but these are not strictly necessary. While it is somewhat difficult to generate mesh topology from scratch using a neural network, [Wang18] proposes a method for generating an object as a mesh from a photograph by deforming an ellipsoid into the desired shape. For many of the examples they show, the network performs admirably. However, the approach as described fails for objects which are not isomorphic to a sphere (for instance, a torus or doughnut), and has a fixed number of vertices, giving it a fixed resolution. Thus, it will oversaturate low vertex-count objects with a number of unneeded vertices, and will have too few vertices to accurately represent objects with high vertex counts.

- *Point Clouds:* Point clouds are the representation format most often used by LiDAR and other forms of depth sensors for storing data about objects. They are low in memory cost, can be used to represent both coarse and fine details, and are already used in object reconstruction in the structure from motion pipeline (for a brief description and history of "structure from motion," see [Granshaw18]). [Fan17] developed a network that takes a photograph as input and generates a corresponding point cloud consisting of 1024 individual points, which they claim is sufficient density to capture the larger-scale structure of the object. However, because a point cloud is just a sampling of locations on the surface of the object, it doesn't convey any information about the surface's topology. This is troublesome, and while there are many methods for converting point clouds into meshes, they are rarely guaranteed to yield closed, watertight objects. Another issue with point clouds is that they are unordered, and so it is difficult to define a single "correct" mapping between the outputs of the network and the desired point set. [Fan17] addresses this second issue with some additional modifications to the network architecture and the loss functions used for training.

- *Voxel Grids:* Voxel grids are the three-dimensional analogue of a pixel-based image, and so they seem a natural choice for object representation. They also interface well with the structure of neural networks – one can design the final layer of the network such that each voxel of the desired output corresponds to one neuron in the output layer. The major issue with this representation format, however, is the large amount of memory required to store the data. For example, a 256x256 image contains 65,536 pixels, whereas a 256x256x256 voxel grid contains 16,777,216 voxels. Additionally, because the sizes of voxels throughout the grid are uniform, they can represent coarse, larger details of the shapes but have difficulty representing finer details. Despite these limitations, [Choy16], [Tatarchenko17], and [Yan17] were able to successfully implement voxel-based image-to-model networks. [Tatarchenko17] partially alleviates the harsh trade-off between the resolution of the voxel grid and the memory needed to represent the object by replacing the uniform grid with an octree data structure – a recursively subdividing data structure that, for most cases, can store voxels much more efficiently than a standard grid. The work presented in [Yan17] is especially remarkable in that unlike most of the previous attempts at AI-based reconstruction, the training procedure does not require ground-truth 3D model data to define a loss function. Instead, the loss function depends on the 2D silhouette projection of the reconstructed geometry. This makes the process self-supervised (the paper refers to it as “unsupervised” but the term “self-supervised” is more accurate).

- *Spline Surfaces:* Spline surfaces are commonly used in the animation and automotive industries, and are included in some form in nearly all major modeling software. They are constructed with a set of control points, which unlike geometric vertices, are not necessarily attached to the object. These control points can be mathematically interpolated to generate a smooth surface with any desired degree of continuity. While there hasn’t been an extensive amount of research into deep learning spline-based object reconstruction, the preprint [Gao19] proposes a model which can generate both spline curves from an input of a 2D line drawing and spline surfaces from an input of a 2D photograph of a 3D object. The paper also describes a useful process for converting a point cloud representation to a spline representation, as the 3D spline surface generation step can take point clouds as input instead of photographs.

There is one overarching problem with all of the object formats described above; they have a fixed resolution when being directly formed from the output of a neural network. While there *are* a few network architectures that can produce different numbers of outputs depending on the inputs (Fully Convolutional Networks, Recurrent Neural Networks, and Transformers to name a few), most networks require that the number of outputs be set to a fixed number once the architecture is defined.

Very recent object reconstruction approaches have tried to solve this problem using *implicit* representations of the geometry as opposed to *explicit* representations.

For the sake of readers unfamiliar with these terms, I'd like to illustrate the difference between explicit and implicit representations with an example. Imagine we wanted to represent the unit ball. Based on the paragraphs above, we can see that one way to do this would be to define approximate vertices for the ball, connect them with edges/faces, and represent it as a mesh. Another way would be to populate the surface with points and define a point cloud. These are *explicit* representations of the object. However, we can also represent the object by defining a function f(x,y,z) that gives information about the object for every point in 3D space. This is an *implicit* representation of the object, and two examples of implicit representation formats, investigated by [Mescheder19] and [Park19] respectively, are occupancy fields and signed distance fields (SDFs). Occupancy fields specify how much an object "occupies" a given location in space, with points inside the object evaluating to one and points outside the object evaluating to zero. Signed distance fields specify how far the surface of the object is from a given location, with a sign to denote whether the point is inside or outside the object. For an occupancy field, the surface of the object is the set of all locations where the occupancy function evaluates to 0.5, and for a signed distance field, the surface of the object is the set of all locations where the field evaluates to 0. See Figure 2.3 for a visual explanation of the differences in these representations.

Both occupancy field and signed distance field representations of objects were successfully learned with deep learning networks in [Mescheder19] and [Park19] respectively. These formats allow for nearly limitless resolution for objects at the cost of a more difficult training setup and an extra step for converting to an explicit representation if one is desired. The network must be trained on the implicit function's values at a wide variety of locations, and so the network needs a massive amount of training examples. Despite these concerns, it is this approach that we will explore in this thesis for converting sketches to approximate 3D models.

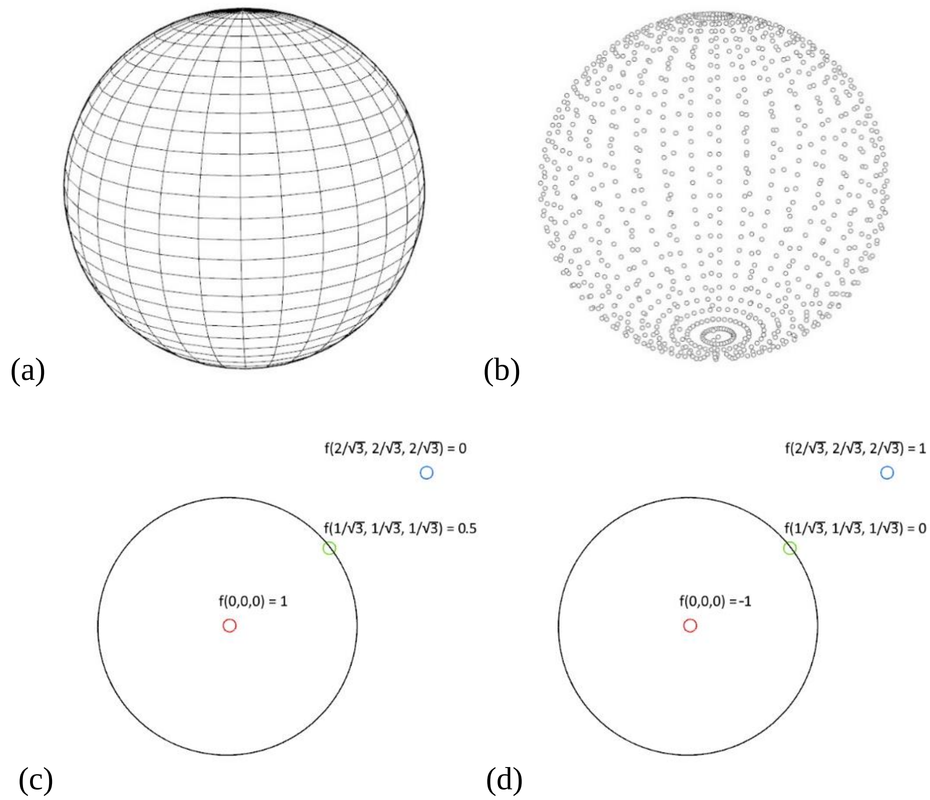


Figure 2.3 Explicit vs. Implicit representations of the unit sphere. (a) A mesh representation of the unit sphere; this is an explicit representation of the object as it explicitly defines the vertices and faces that form the geometry of the object. (b) A point cloud representation of the unit sphere, this is an explicit representation of the object as it explicitly defines a set of points that make up the surface of the object. (c) An occupancy field representation of the unit sphere; this is an implicit representation of the object as it defines a function across the entire 3D space that specifies the relationship between a point and the object - in this case, whether the volume of the object occupies the point. (d) A signed distance field representation of the unit sphere; this is an implicit representation of the object as it defines a function across the entire 3D space that specifies the relationship between a point and the object - in this case, the distance between the point and the object surface with a sign to indicate whether the point is inside the object. In c) and d), I show the evaluation of these implicit representations at three locations in space along with the outline of the surface of the object.

Latent Space Studies:

As stated in the previous chapter, we will be trying to develop a latent space for the sketch-to-model problem. While there is a great deal of research on latent spaces, I'd like to focus on two particular papers that developed latent spaces for both halves of the sketch-to-model problem: one for sketches and one for 3D models. Both papers rely on variational autoencoders, a type of network presented in [Kingma14] and used for this thesis that produces smooth latent space encodings.

The first selected paper is [Ha17], which trained a recurrent neural network on a stroke-based dataset created by human users to develop latent spaces for various categories of sketches. The latent spaces for their experiments had 128 dimensions, and they were smooth in the sense that one could interpolate between latent vectors and obtain reasonable sketches when the interpolated vectors were decoded back into the original format (see the top of Figure 2.4). Additionally, when a sketch outside of the category covered by a given latent space was encoded into the space and then reprojected back to a sketch, the result was a new sketch that fit the category of the latent space and still resembled the original sketch (see the bottom of Figure 2.4 for an example). This behavior could be extremely useful in the sketch-to-model problem; we would ideally like our network to take in shaky, rough line drawings as input and remove the inaccuracies in the objects it generates. If the latent space only encodes flat, smooth models, then by projecting a sketch into the latent space before reconstructing it, we can remove its imperfections.

The second paper on latent spaces is [Wu16], which trains a generative adversarial network (GAN) to generate voxel representations of 3D objects while learning a latent space for these representations. This latent space had 200 dimensions and was used to generate voxel representations within a 64^3 resolution grid. After learning the latent space, the network could be used for classification and image-to-model reconstruction with reasonable quality results.

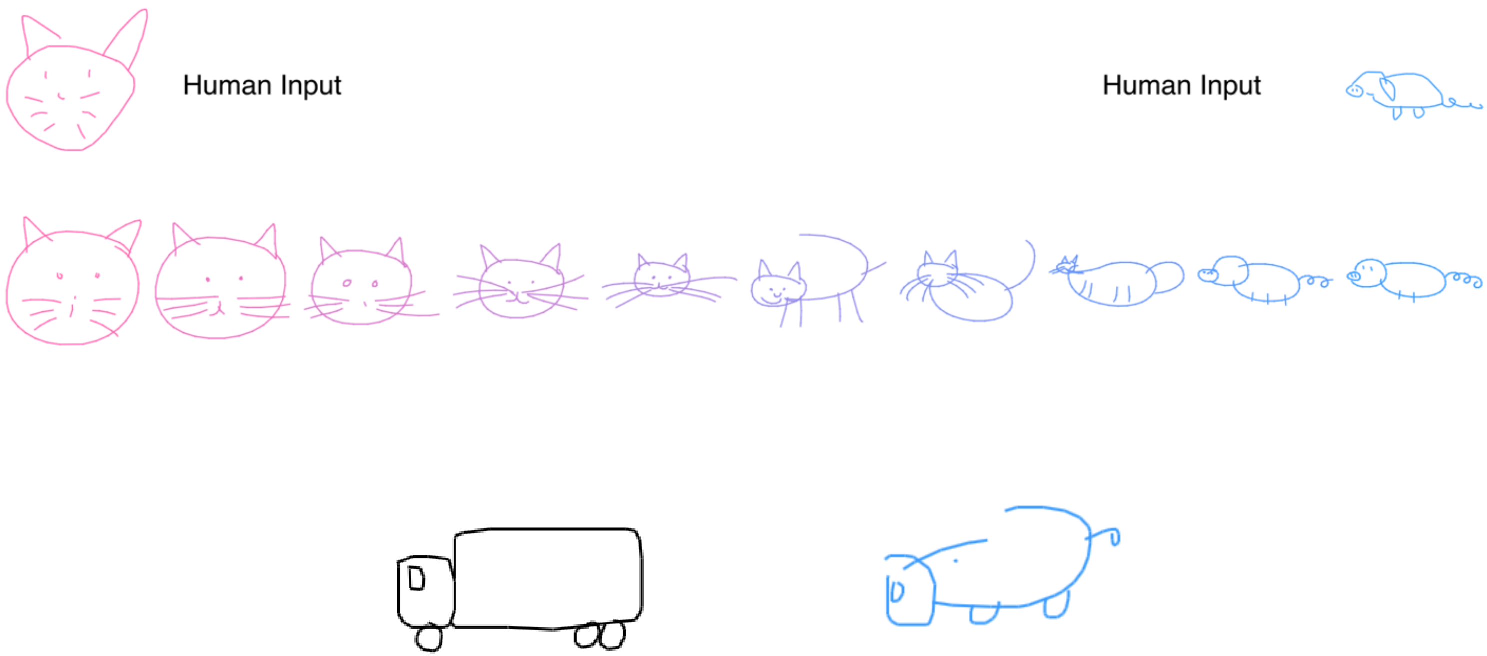


Figure 2.4 Selected figures taken from [Ha17][1]. (Top) An interpolation between the encoding for a cat sketch and a pig sketch, created by interpolating the latent vectors of the inputs and decoding the interpolated results. Note that all of these interpolation steps decode to reasonable sketches, and that the transition from the cat sketch to the pig sketch is relatively smooth. (Bottom) An encoding of a truck sketch into a latent space trained only on pig sketches. The result is a pig that approximately resembles the original truck.

[1] The original figures are copyrighted by David Ha and Douglas Eck, the authors of the original paper. They are included here under the premise of 'fair use' per Section 107 of the Copyright Act.

Sketch-to-Model Studies:

Finally, I would be remiss not to include examples of machine learning approaches that address the sketch-to-model problem itself. [Zhang21] demonstrates incredible promise with view-aware model generation from sketches, though the method's good performance does seem to be limited to the categories of objects available in the ShapeNet training dataset. [Delanoy18] addresses this issue by training on both existing dataset examples taken from the ShapeCOSEG dataset as well as examples created with automated Constructive Solid Geometry (CSG) modeling. The approach presented in [Delanoy18] resembles the sketch-to-model process presented in this thesis work with two major differences. The first difference is that [Delanoy18] uses a single convolution neural network for sketch-to-model reconstruction, with skip connections to preserve high-resolution information at deeper layers of the network. Conversely, in this thesis, the sketch-to-model process was split into two networks, where the first was specifically tasked with learning latent representations for sketches, and the second was tasked with reconstructing geometry from these latent representations. The second difference is that the network developed in [Delanoy18] outputs a voxel grid representation rather than an implicit representation, and this format limits the resolving capabilities of the network and drastically increases the data requirements as discussed in the "Voxel Grids" bullet point above.

Chapter 3

Dimensional Analysis of Sketches

As stated previously, one of the goals of this thesis is to develop a latent space for facilitating the reconstruction of 3D objects from sketches. [Park19] as well as other papers discussed in the last chapter learn the latent space using the 3D objects themselves, but I see this as problematic for reasons that are illustrated in Appendix F. Thus, for this thesis, I sought to use machine learning to develop a latent space from 2D sketches of 3D objects rather than from the 3D objects themselves.

One of the crucial pieces of information for understanding sketches in a machine learning context is some sense of the “dimensionality” of sketches. Below I present a few useful techniques to identify the number of dimensions needed to represent sketches and to reduce sketches to those lower-dimensional representations.

3.1 Background

The overwhelming majority of machine learning techniques require an input to be represented as a set of features. These features are typically binary, discrete but non-binary, or real-valued, and we define the set of all possible combinations of values for these features as the *feature space*. We refer to the number of features as the *dimensionality* of the feature space, and this provides a naïve upper bound on the complexity of the machine learning problem. There are two main reasons why this upper bound cannot be completely trusted. First, if the feature space doesn't contain all the information needed to learn the problem, it cannot guarantee an upper bound on the complexity. Second, if the feature space contains redundant or unnecessary information, it provides a loose upper bound on the complexity that can be tightened. For an example machine learning problem to help explain these concepts more thoroughly, please see Appendix A.

Returning to the problem of interpreting sketches, a natural question to ask at this point would be: "What should we consider to be the feature space for sketches?" One answer to that question is this: if we restrict our definition of "sketches" to being grayscale, rasterized images, then we can consider each pixel in the image to be a feature. Continuing with this logic, if we choose to consider images with 8-bit color (grayscale) resolution and 256 pixels by 256 pixels spatial resolution, we will have 65,536 features in our feature space, where each feature can have 2^8 possible values. With these assumptions, the complexity of our machine learning problem can be upper-bounded as a 65,536-dimensional problem.

In some sense, it is humbling to define an upper bound on the creative possibilities of human sketches in this way, but considering that the number of possible images given these constraints is $2^{524,288} \approx 2.6 \times 10^{157,826}$, there's still plenty of room for "creative liberty." To give some context for how enormous this number truly is, there are estimated to be on the order of 10^{78} to 10^{86} atoms in the observable universe with the "back of the envelope" calculations outlined in [Villaneuva09]. Thus, if you were to attempt to sketch this many images on paper with a real pen, and all other atoms in the universe were converted into the necessary ink, you wouldn't even make a dent in the number of possible sketches before running out of ink, even if it only took 1 atom of ink per sketch.[2]

I'd like to take a moment now to quickly discuss why the 65,536-dimensional bound might not be an accurate representation of the complexity of the set of possible sketches. There are two arguments I'd like to explore: first, the argument that the bound is too small, and second, the argument that the bound is too large.

For the first of the two arguments, it can be easily shown that many sketches are significantly more complex than what a 256x256 resolution image can depict. Many skilled artists' sketches have both coarse and fine details, and those fine details often cannot be represented with limited image resolution. An admittedly contrived example of this is shown in Figure 3.1. The general structure of the sketch can still be seen with 256x256 resolution, but the small ridges at the top of the lower walls are lost with the reduced resolution. Thus, we cannot claim that all possible sketches can be represented by a 256x256 image, or by any fixed resolution image for that matter, as we can always add finer details that exceed the sampling rate of the chosen image resolution. Despite this limitation, if we adjust our machine learning problem to "converting sketches *which can be represented by 256x256 resolution images* to three-dimensional models," we can still use 65,536 dimensions as an upper bound for the number of dimensions needed to represent these sketches.

[2] The author is well aware that ink is a mixture and therefore there is no such thing as an "atom" of ink. However, it is the author's opinion that incorporating the minimum number of atoms required for a single "unit" of ink into the calculations for this statement would add unnecessary complications and would therefore take away from the point being made.

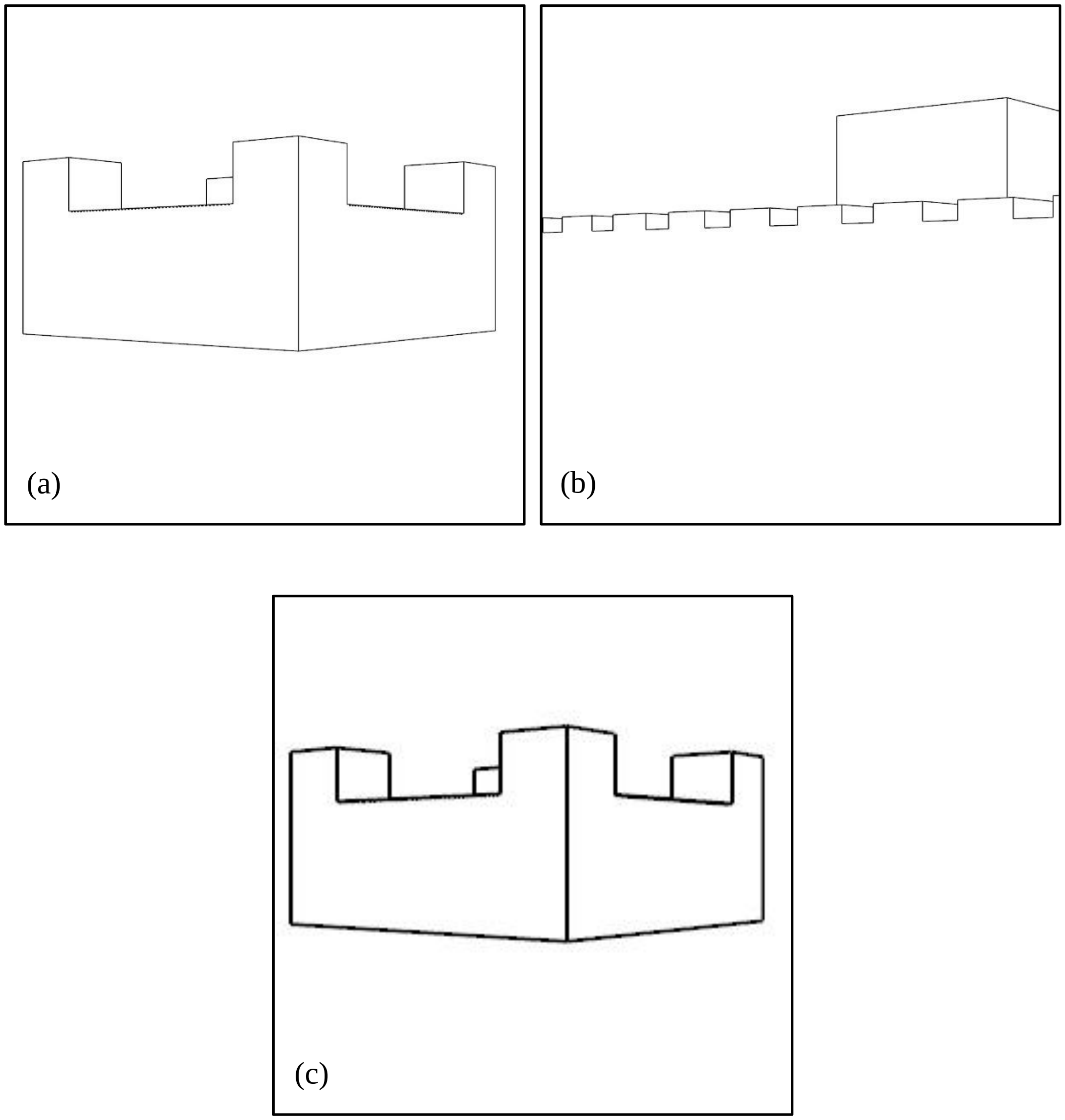


Figure 3.1 An example of the resolution limits of the rasterized format. (a) A higher resolution sketch of a simple castle-like structure with small ridges at the top of the lower walls. Even at this resolution, the fine details are difficult to discern. (b) A zoomed in version of the ridges on the top of the lower walls. (c) A 256x256 rasterization of the sketch; note that the ridges are effectively impossible to recognize at this resolution.

For the second of the two arguments, it is fairly easy to conclude that a vast number of images in the set of all possible 256x256 resolution, 8-bit per pixel images are not sketches. Below are three images that were generated by choosing a random value for each of the 65,536 pixels.

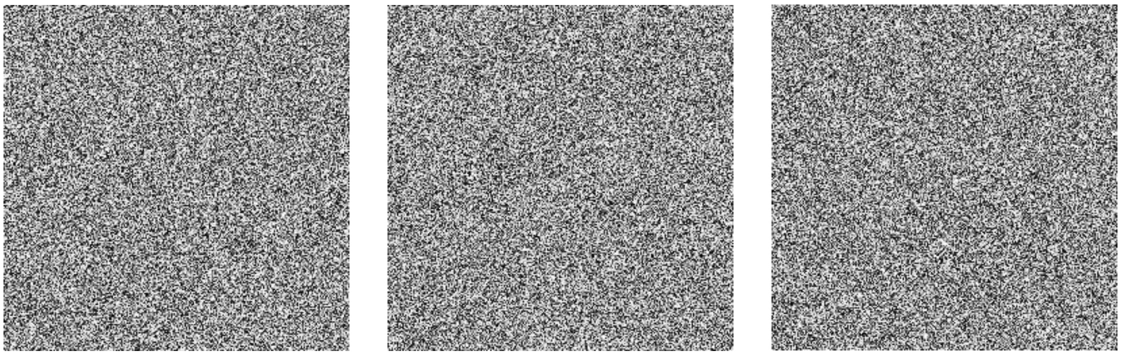


Figure 3.2 Three images created by randomly generating each pixel from the set of all 2^8 possible values. Despite these three images differing significantly from one another quantitatively, qualitatively they do not appear significantly different from one another, nor do they appear to convey any kind of meaning.

While one may be able to identify some interesting patterns that arise from the chaos, it would be hard for anyone to argue that these images are sketches. In reality, the overwhelming majority of the images in this pixel-based feature space are like those shown above, and so there is a strong argument to be made that the true dimensionality of the set of sketches is significantly smaller than our current bound of 65,536. Moreover, because the task at hand is the reproduction of 3D geometries from their 2D sketches, we need only concern ourselves with the set of images that convey 3D geometry, which is an even smaller set than the set of images that can be considered sketches. Intuitively, this should reduce the dimensionality of the problem even further.

What, then, can we do to more concretely determine the true number of dimensions needed to represent our input sketches? One way would be to find a different feature space with fewer dimensions that is still capable of representing sketches. We could then use the dimensionality of this new feature space as a tighter upper bound on the complexity of sketches, and take advantage of this new space to improve our machine learning algorithm.

Before I discuss how to find lower-dimensional feature spaces, I'd like to give a quick overview on *reprojection error*, a metric for evaluating the "accuracy" of a lower-dimensional feature space. Although ideally a lower-dimensional feature space would retain all the information present in the higher-dimensional feature space, there is often a loss of information when a higher-dimensional representation is *encoded* in a lower-dimensional feature space. One way to identify this lost information is to *decode* the lower-dimensional representation back to the original higher-dimensional feature space and to measure the difference between the original sample and the decoded result. This difference is called the *reprojection error,* and it is illustrated in the simple case of reducing from two dimensions to one dimension in Figure 3.3.

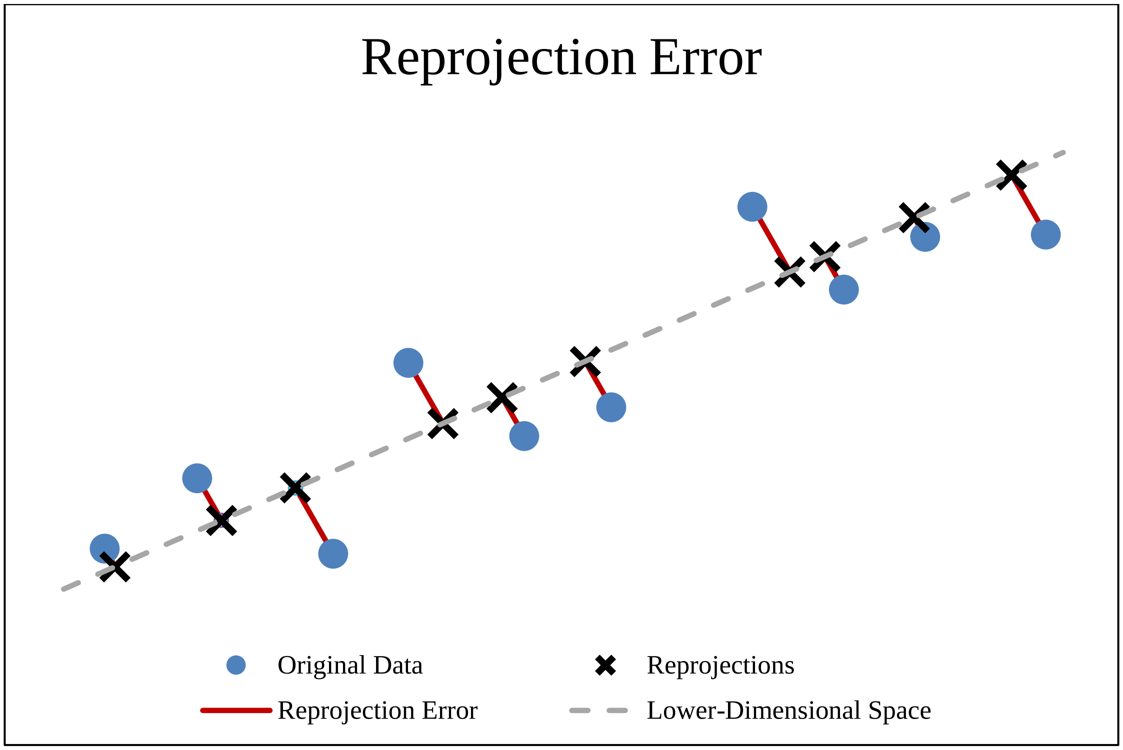


Figure 3.3 An illustration of reprojection error. The original, higher-dimensional data is shown as blue circles, and the lower-dimensional feature space is shown as a dashed grey line. When the 2D dataset is encoded into the lower-dimensional feature space, it is projected to the dashed line and reduced to one dimension. The black X marks on the line represent the decoded reprojections, and the solid red lines between each datum and its reprojection represent the reprojection errors.

With this tool in hand, we simply need to find candidates for possible lower-dimensional feature spaces and evaluate the reprojection errors of example sketches to determine the spaces' performance. In the next section, I will cover Principal Component Analysis (PCA), a commonly used technique for identifying lower-dimensional representations of data.

3.2 PCA of Sketches

One of the most well-known methods for identifying lower-dimensional representations of labeled or unlabeled datasets is Principal Component Analysis (PCA). A full explanation of PCA from base principles is beyond the purview of this work, but I do explain the process briefly in Appendix B. Please see the appendix for more information.

PCA has been successfully used in a variety of wide-ranging applications. One of the earliest and most interesting success stories of PCA in the context of images is in facial recognition. [Turk91] introduced the concept of using “Eigenfaces” for representing faces for facial recognition; these “Eigenfaces” are the principal components of the images of faces found within a dataset. The authors of the paper used PCA to find the Eigenfaces of the dataset and projected the original dataset images into the lower-dimensional “Face Space” made by these Eigenfaces. Then, for any new face image, they found the nearest neighbor from the dataset in the “Face Space” and classified the new face image as being the same person as the nearest neighbor, or “unknown” if the nearest neighbor was too dissimilar to the new image.

Just as Turk and Pentland applied PCA to find “Eigenfaces” for facial recognition, we can use PCA to find “Eigensketches” for our dataset and project sketches to their “Sketch Space” representations. If these representations can be accurately reprojected within a reasonable margin of error, then we can use the “Sketch Space” representation to reduce our machine learning problem complexity. Furthermore, because PCA finds all the principal components of a dataset, we can tune the number of Eigensketches we include in the Sketch Space to find a desired trade-off between accuracy and dimensional reduction.

Two independent datasets of sketches were generated automatically with a script written in Grasshopper, a visual scripting software that ships with McNeel's Rhinoceros3D modeling software. The script randomly generates three boxes in a predefined region and performs random Boolean operations (union or difference) on them to create one compound object. These boxes are then rendered as line drawings and saved to a directory. Examples of the generated sketches are shown in Figure 3.4.

The first dataset contains 7795 sketches, and the second contains 4879 sketches. The reason for generating two separate datasets was to examine the variability between the principal components with the hope that they were similar or identical for both datasets. The actual implementation of PCA was done using the scikit-learn Python library, and required each image to be flattened into a 65,536-dimensional vector. The resulting principal components can then be unflattened back to a 256 by 256 image for ease of viewing. The first three components for both datasets are shown in Figure 3.5.

While these images may not seem to resemble sketches at all, by combining a number of them together in various combinations, one can approximately recreate the sketches of the dataset. Unfortunately, we can see that by the third principal component, the two datasets' analyses have deviated from one another. This indicates that the components found with PCA are highly specific to the dataset, and will not generalize to a larger body of sketches.

Examples of sketches and their reconstructions with various numbers of principal components are shown in Figure 3.6. With 50 principal components, the basic shape of the sketch can be captured, and with about 500-1000 principal components, most of the details can be recreated, albeit with minor distortions.

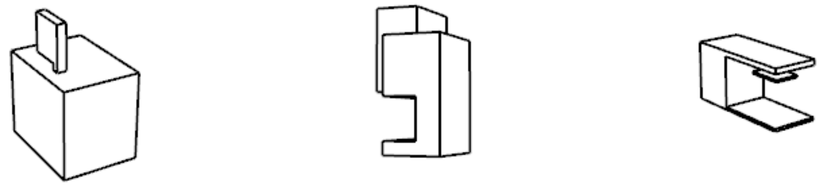


Figure 3.4 Example sketches from the first dataset used for PCA.

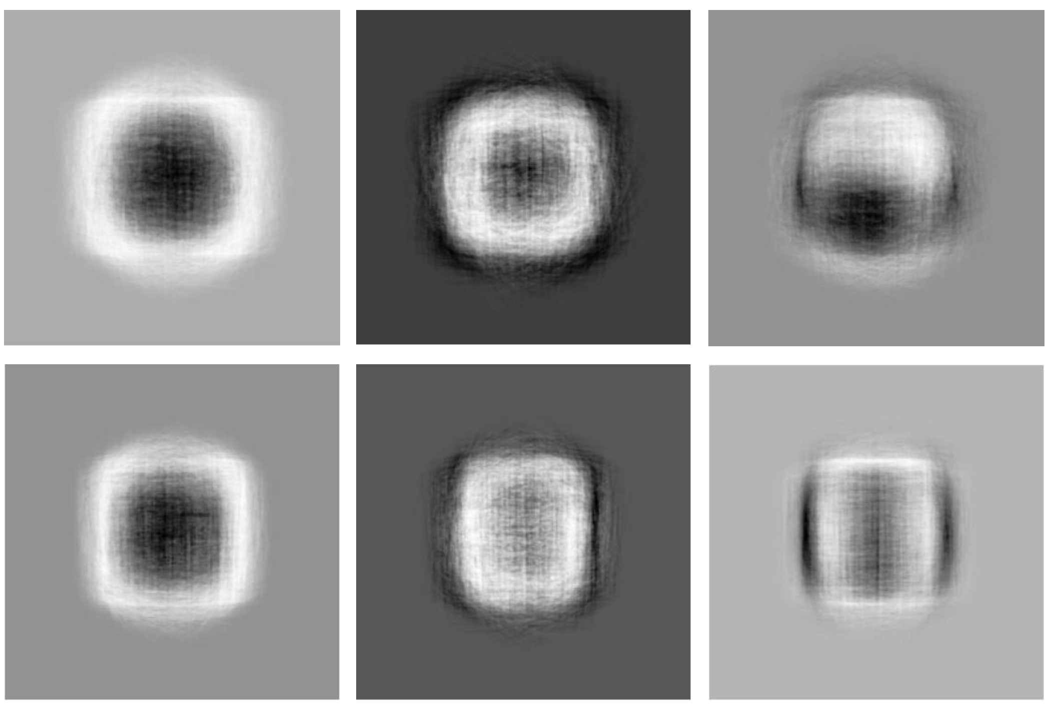


Figure 3.5 The first three Eigensketches of both datasets, visualized with PyPlot. The Eigensketches of the first dataset are in the top row, and the Eigensketches of the second dataset are in the bottom row. Even by the second Eigensketch, the datasets start to show noticeable differences, and by the third Eigensketch, the two datasets have entirely different principal components. Note that the PyPlot visualization normalizes the images to their minimum and maximum values, so the exact colors shown above are less significant than the relative color differences within each image.

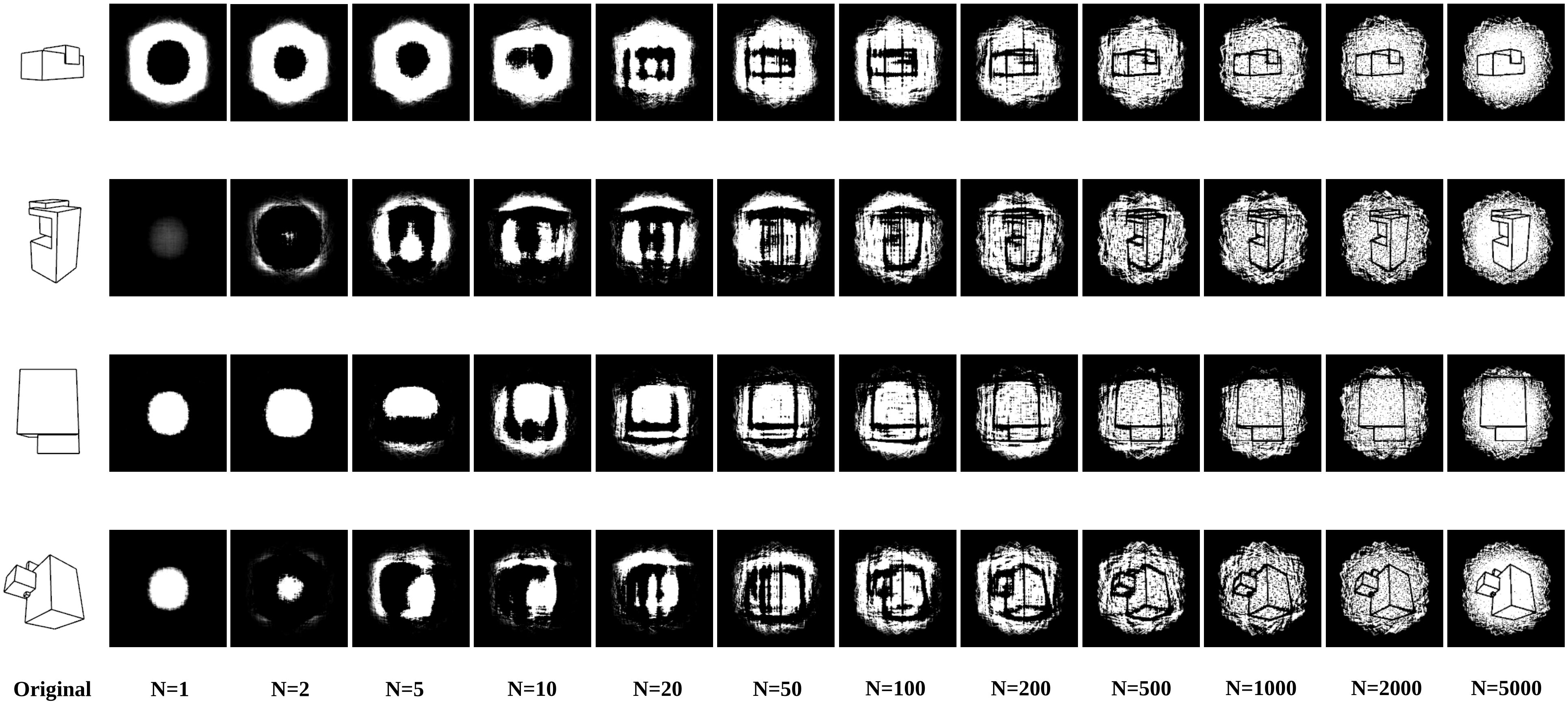


Figure 3.6 Example set of sketches and their reprojections for various numbers of principal components. 'N' in the figure above represents the number of principal components included in the reprojection. As can be seen, increasing the number of components increases the visual quality of the reprojections. Best viewed digitally.

To quantitatively determine the “ideal” number of dimensions for our low-dimensional representation, we can look at a plot of the eigenvalues of the principal components and choose a cutoff point where the eigenvalues are too small to greatly influence the projection. The eigenvalues of the principal components of the two datasets are plotted at the top of Figure 3.7. Visually, the plotted curves are nearly identical and they tend to zero very quickly, indicating that the number of principal components needed to accurately represent the sketches is largely independent of the specific dataset (even if the components themselves are heavily dependent on the dataset).

Based on the eigenvalue plots at the top of the figure, we might be tempted to choose an early cutoff point near the “elbow” of the curves shown at the top of Figure 3.7, say somewhere around 200 principal components. However, this would be a mistake, as we can see in Figure 3.6 that the 200-component case is highly distorted and thus loses a large amount of the desired information. The issue is that the first few principal components have such large corresponding eigenvalues that the differences between the eigenvalues of the remaining components are difficult to compare. To visualize the smaller eigenvalues more clearly, at the bottom of Figure 3.7, I show the same plot as before but with a logarithmic scale for the vertical axis. Looking at this new plot, it is clear that the eigenvalues still decrease significantly in regions where the original plots looked almost entirely flat.

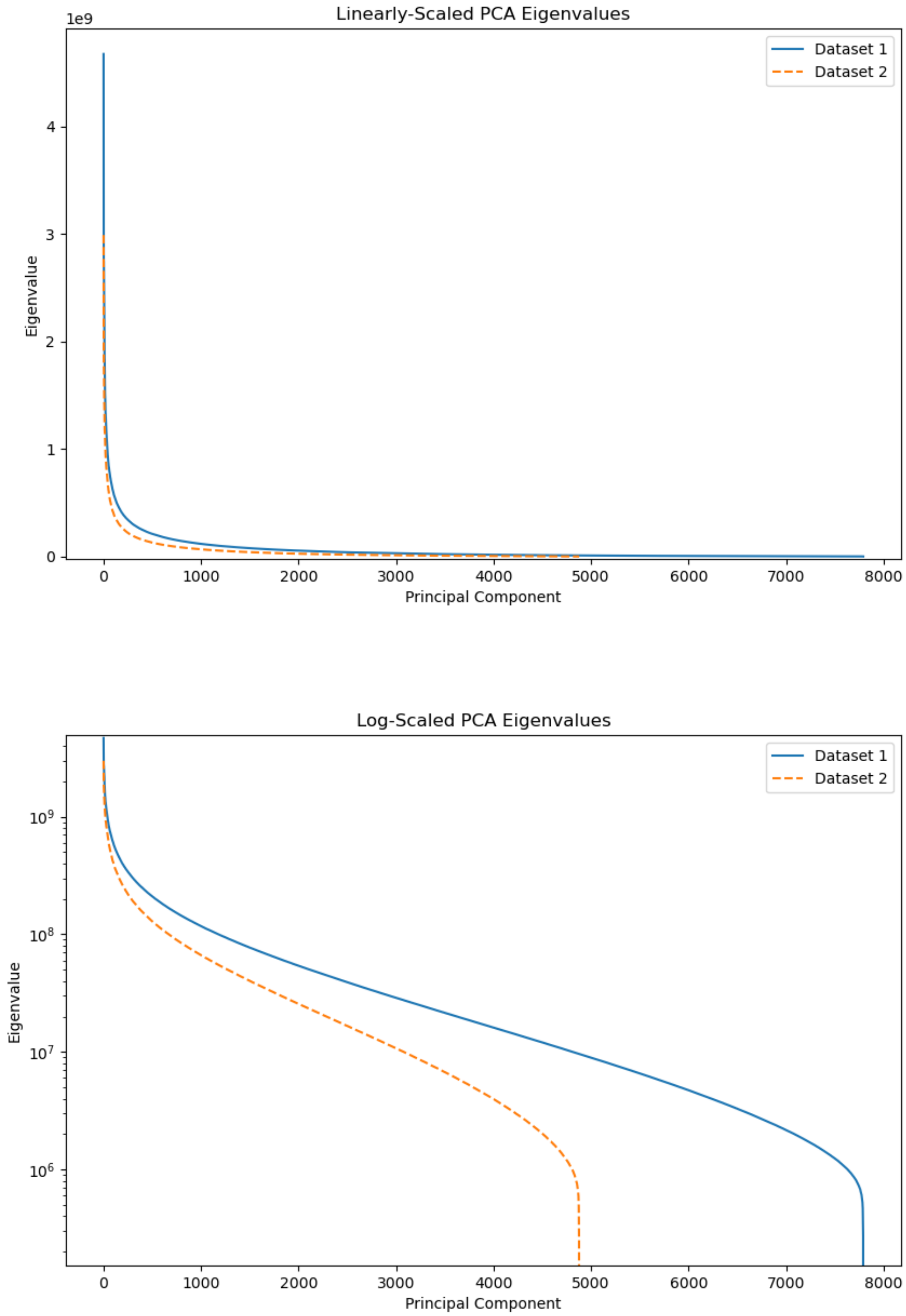


Figure 3.7 The PCA eigenvalue plots for the two datasets with a linearly scaled y-axis (Top) and a logarithmically scaled y-axis (Bottom). The differences between the eigenvalues of the two datasets' later principal components is much more apparent with the logarithmic scaling.

Why is it that the first few principal components seem to dominate the others despite not having nearly enough information to represent the object completely? The reason is somewhat complex, but it stems from the fact that PCA relies on Euclidean distances in the feature space to find the "best" representation, which in the context of our sketches, is essentially a measure of the differences at each pixel. However, these pixel differences do *not* correspond to the types of "meaningful" differences between the sketches that human observers would identify. To help illustrate this concept, consider the three sketches in Figure 3.8.

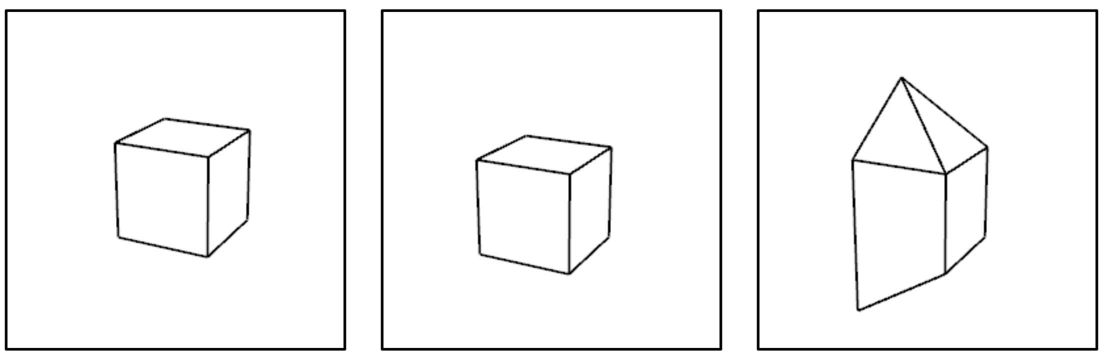


Figure 3.8 Three sketches of 3D objects. While humans consider the left and middle sketches to be the most similar, the principal component representations of the left and middle sketches are nearly twice as different as those of the middle and right sketches, as shown in Table 3.1.

As human observers, we would consider the two leftmost sketches to be the "closest" in semantic meaning. These two sketches are nearly identical - both contain the same lines in the same orientation, just slightly translated. Both would be considered sketches of cubes.[3] The rightmost sketch, however, is significantly different from the other two. It has additional lines not found in either of the other two sketches and lacks

[3] Because this is a perspective drawing, albeit one with very little perspective distortion, a simple translation will slightly alter the geometry of the 3D object being represented. The difference is slight enough that it is well beneath the expected "margin of error" for sketches, and so most human observers would still interpret this as a cube.

a few of the lines common to the other sketches. Moreover, unlike the first two sketches, it is not a sketch of a cube; it is a sketch of an irregular polyhedron.

Unfortunately, these similarities are not reflected in the per-pixel differences between the sketches. The norm[4] of the pixel differences between the leftmost two sketches is roughly 1.43 times *larger* than the norm of the differences between the middle and right sketches. This issue is even more pronounced for the Eigensketch representations; the norm of the difference between the left and middle sketches is more than twice that of the middle and right sketches, and the largest of the three difference norms is that of the difference between the two sketches of cubes. Thus, we can conclude that the Eigensketch representation is not an effective low-dimensional representation for the sketches. The norms of the pixel differences and Eigensketch differences are shown in Table 3.1.

Sketches Compared	**Pixel Difference Norm**	**Eigensketch Difference Norm**
Left & Middle	38.376	23.273
Left & Right	41.355	22.895
Middle & Right	26.784	10.232

Table 3.1 A summary of the magnitudes of the pixel differences between each possible pair of images in Figure 3.8. Despite their visual similarities, the left and middle sketches are nearly twice as different as the middle and right sketches in the Eigensketch representation.

[4] For those familiar with the term 'norm,' the type of norm used here is an L2 norm. For those unfamiliar with the term, the norm here is computed with a higher-dimensional analogue of the Pythagorean Theorem – the difference between the two images is computed in each dimension of the feature space, these difference values are squared, all of them are summed, and then the square root of the sum is the norm. The norm is therefore the higher-dimensional analogue of the hypotenuse from the standard Pythagorean Theorem.

There are a few ways to try and adjust PCA to suit our needs (please see Appendix B for more information), but these adjustments raise considerable problems of their own, namely increasing the size of the feature space dramatically for the analysis. One of the unfortunate side effects of this increase in dimensions is that the resulting components are even more likely to overfit to our datasets, an issue we already have with the standard PCA. Thus, it would seem that a new approach is needed, and that new approach is the focus of the next chapter.

Chapter 4

Learning a Sketch-Based Latent Space

We saw in the last chapter that one of the standard dimensionality reduction techniques, PCA, did not suit our purposes, as the assumptions of the underlying mathematics fail to capture meaningful information about the sketches. Here, we take a different approach to the same problem. In this chapter, we attempt to *learn* a latent space from a dataset of 2D sketches using a variational autoencoder.

4.1 Variational Sketch Encoding

Learning lower-dimensional representations of higher-dimensional data is not a new problem, and the networks that do so are usually referred to as autoencoders. For more information about autoencoders and the technical specifics relevant to this thesis, please see Appendix C. Based on the nature of sketches as an image input, I concluded that a convolutional variational autoencoder would best serve my purposes. Notably, the convolutional layers of the network allow for the autoencoder to be shift-equivariant, which will help to address the image translation issues raised at the end of the last chapter.

I tested many different network architectures, but they all shared the same overall format: an encoder consisting of a set of convolutional input layers followed by one or more fully connected layers, and a decoder with the same layers as the encoder but in reverse order. Note that the final fully connected layer for the encoder contains twice the number of neurons as the desired number of latent dimensions; this is because variational autoencoders form latent representations with Gaussian distributions rather than with discrete values, and a Gaussian distribution requires two values - usually a mean and a variance - to define. The standard TensorFlow tutorial for constructing a convolutional variational autoencoder recommends an encoder architecture with two convolutional layers and one fully connected layer, and I used this as a baseline for my initial tests.

My first set of network architectures contained two to six convolutional layers and one fully connected layer. The first convolutional layer had 32 filters, and all subsequent layers had double the number of filters of the previous layer. The size of the filters in the first two layers was 5x5, and in all additional layers was 3x3. Each convolutional layer had a stride of two, so the size of the layer's output was reduced by roughly a factor of two for each additional layer (double the number of channels times one-fourth the number of "pixels" results in half the amount of data). All encoder convolutional layers used 'valid' convolution, whereas all decoder transposed convolutional layers used 'same' convolution. All convolutional layers had ReLU activation functions. The fully connected layer converted the final convolutional layer's output to the latent representation, and the set of latent dimensions used was {512, 1024, 1536, and 2048}. Thus, there were a total of 20 different network architectures in the initial set of runs.

The networks were trained on the 7795-sketch dataset used for the PCA analysis in the previous chapter. The dataset was split into training and testing sets where 90% of the examples were used for training and 10% were used for testing. The networks were trained using an ADAM optimizer (see [Kingma15] for more information) with a learning rate of 5×10^{-5}. The batch size for the training/testing sets was 40 sketches. The training hardware was an NVIDIA GeForce RTX 3090 GPU, and it took roughly 14-17 seconds per epoch. The shallower networks stopped learning after fewer epochs, and so the total number of epochs used for training these networks was reduced to avoid pointless computation. Tables 4.1 and 4.2 summarize the major details of the initial training runs. Figure 4.1 shows a plot of the test set Evidence Lower Bound (ELBO) over the course of training for all architectures, and Figure 4.2 shows a visual comparison between an example dataset sketch and its reprojection from the latent space for the best run.

Network Training Epochs	
Network Architecture	**Epochs**
2 Convolutional Layers	100
3 Convolutional Layers	100
4 Convolutional Layers	150
5 Convolutional Layers	150
6 Convolutional Layers	160

Table 4.1 Total number of epochs used for each architecture.

Network Performance (Maximum ELBO across all Epochs)				
Network Architecture	**Latent Dimensions**			
	512	*1024*	*1536*	*2048*
2 Conv. Layers	-3027	-3019	-2981	-2951
3 Conv. Layers	-2776	-2687	-2651	-2635
4 Conv. Layers	-2501	**-2369**	-2388	-2391
5 Conv. Layers	-2468	-2466	-2500	-2486
6 Conv. Layers	-3126	-3193	-3203	-3180

Table 4.2 Maximum ELBO for each architecture-latent space pair of the initial training runs. Note that because the ELBO metric is negative, the maximum ELBO is the lowest magnitude value.

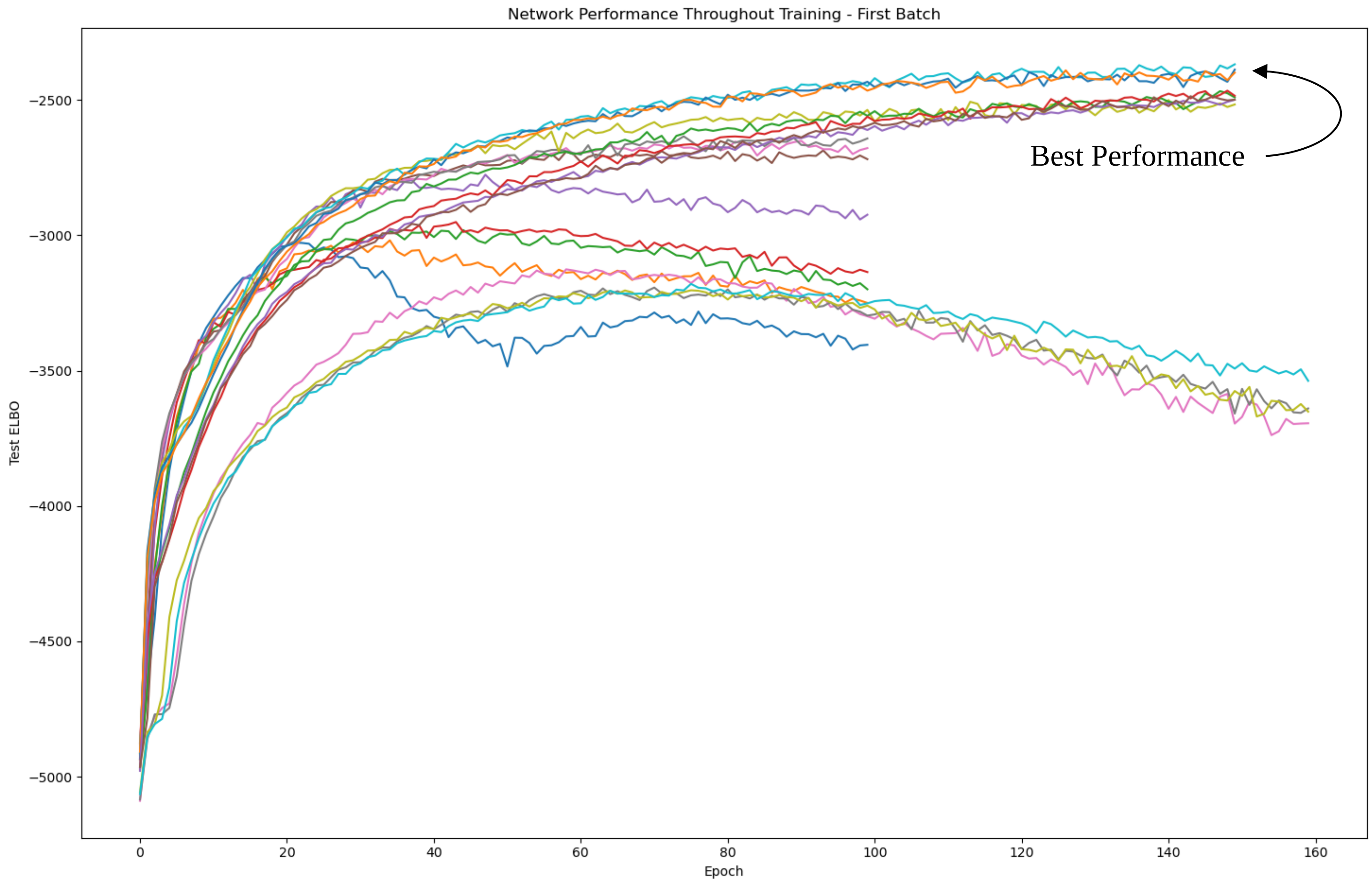


Figure 4.1 Test ELBO over the course of training for the first set of network architectures. The best performing architectures were those with 4-convolutional layers, as they had the least negative ELBO value. Note that some of the runs were stopped earlier than others as stated in Table 4.1.

Figure 4.2 A visual comparison of an example dataset sketch and its reprojection from the learned latent space. The reprojection maintains the overall structure of the original sketch, but smaller details are lost.

For a roughly 65x reduction in the information, the performance of the network is admirable. However, it is still clearly missing important information about the model – some of the edges are missing, and the smaller protrusions have been blurred beyond recognition. One thought to try and alleviate this issue was to add additional fully connected layers at the end of the encoder/start of the decoder to allow for additional processing of the input. To test this idea, two networks were constructed using the best performing network architecture as a starting point (4 convolutional layers and 1 2048-neuron fully connected layer). To this base network, I added additional fully connected layers after the convolutional layers. Because these new network architectures were more complex than those of the initial runs, I increased the number of epochs allotted for training up to 200, and the performance plots show that the networks stopped improving well before the total number of epochs was reached. Table 4.3 provides a summary of the performances of these networks, Figure 4.3 shows a visual comparison of a test image and its reprojection, and Figure 4.4 shows an example training plot of the best performing network. Unfortunately, these networks did not perform any better than the best performing network of the first set, both in terms of ELBO and visual comparison of the test images and their reprojections, and so I concluded that I would need to find another method to try to improve the reconstruction.

Network Performance (Maximum ELBO across all Epochs)		
Additional Network Layers	**Latent Dimensions**	**Maximum ELBO**
1 1024-neuron fully connected layer	1024	-2692
2 1024-neuron fully connected layers	1024	-2963
1 1536-neuron fully connected layer	1024	**-2603**
1 2048-neuron fully connected layer	1024	-2609

Table 4.3 A summary of the second set of training runs, including the specific architectures tested and their test performances. The best network performance of this set of runs was that of the network with one additional 1536-neuron fully connected layer, but it was quantitatively much worse than the performance of the best network from the initial set of training runs.

Figure 4.3 A visual comparison of an example test image and the test image's reprojection from the latent space. The distortions are slightly worse here than in the reprojection shown in Figure 4.2.

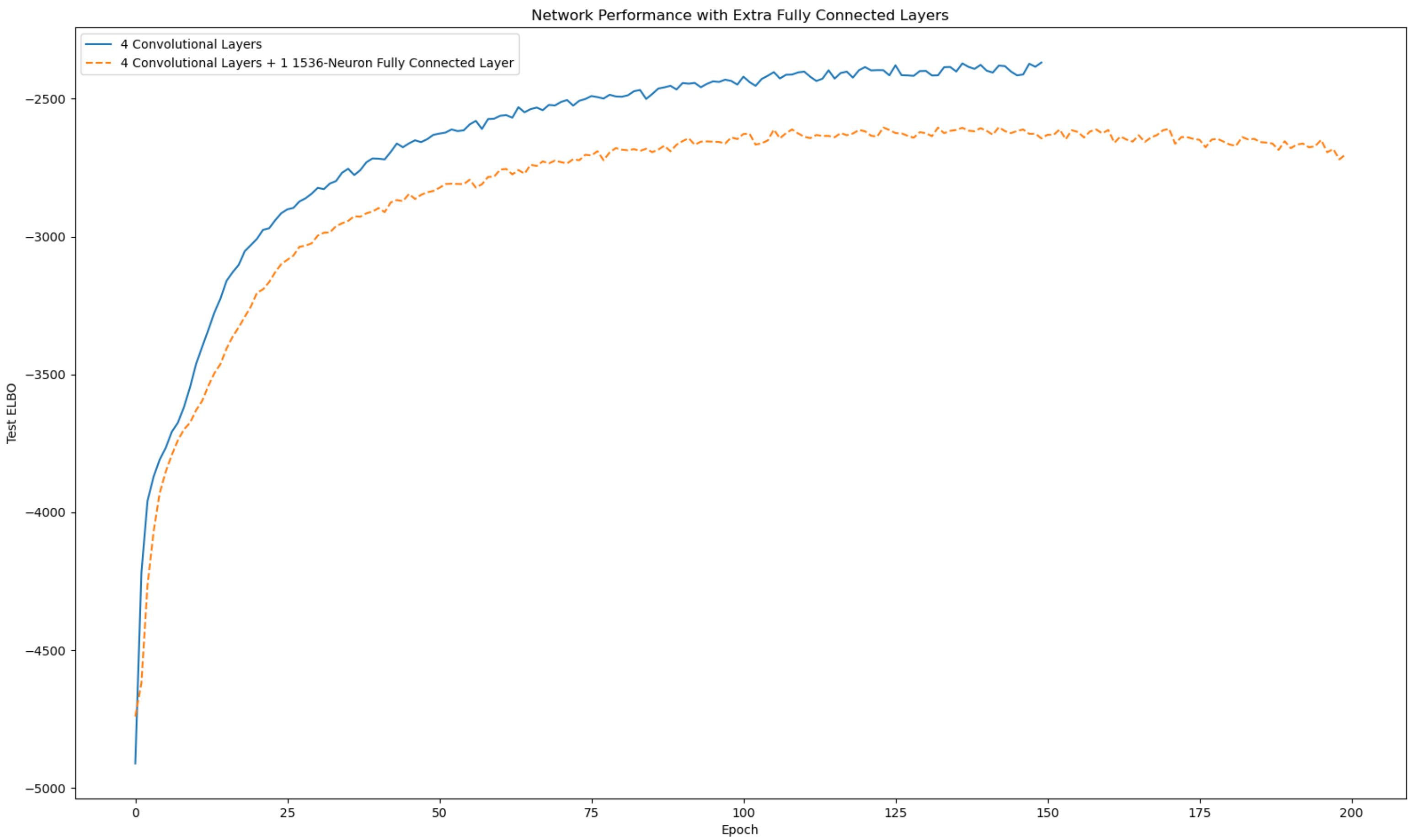


Figure 4.4 Test ELBO over the course of training for the highest performing network of the second set compared with the highest performing network from the first set. The performance curves are similar, but the performance of the network with the additional fully connected layer plateaus at an ELBO value roughly 250 less than the highest performing network from the initial runs.

4.2 Data Augmentation

Algorithms for identifying lower-dimensional representations of high-dimensional data are often prone to overfitting, as discussed in both the previous chapter with respect to PCA and in Appendix C with respect to standard autoencoders. While overfitting can be alleviated to some extent by adjustments to the learning algorithm, it is much more important to ensure that the algorithm has a large enough training set to learn the problem. It is exceedingly difficult to tell the difference between meaningless patterns specific to the dataset and meaningful lower-dimensional structure of the set of possible inputs for the learning problem. The more examples there are in the dataset, the less likely it will be that the examples follow meaningless patterns. While it was possible to generate more examples for this work, as the generation of the sketches was done automatically, increasing the size of the dataset would be much more difficult for a dataset of user-drawn sketches. Even though I did not have access to such a dataset for this thesis work, I would eventually like to apply the techniques developed here to a hand-drawn dataset, and as such, I decided to find a solution for increasing the available training data that was feasible for both synthetic and hand-drawn sketch datasets: data augmentation.

For readers unfamiliar with the term, data augmentation is a process wherein training examples for a machine learning algorithm are programmatically altered to provide the learning algorithm with additional examples. For example, a preprocessing step could randomly choose whether to flip the input image horizontally before feeding it into the learning algorithm, and this would double the number of possible training examples. Another preprocessing step could randomly rotate an image by some angle before feeding it into the learning algorithm, and depending on the number of possible angles, this could drastically increase the number of possible training examples. Other types of augmentation include translation, scaling uniformly or anisotropically, recoloring, random per pixel perturbations (adding noise), etc. A few visual examples of data augmentation are shown in Figure 4.5.

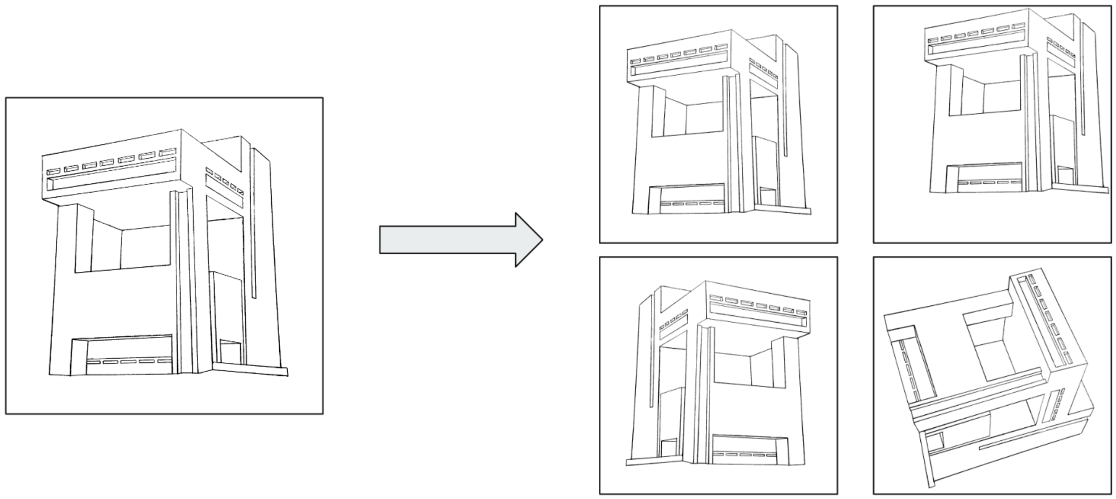


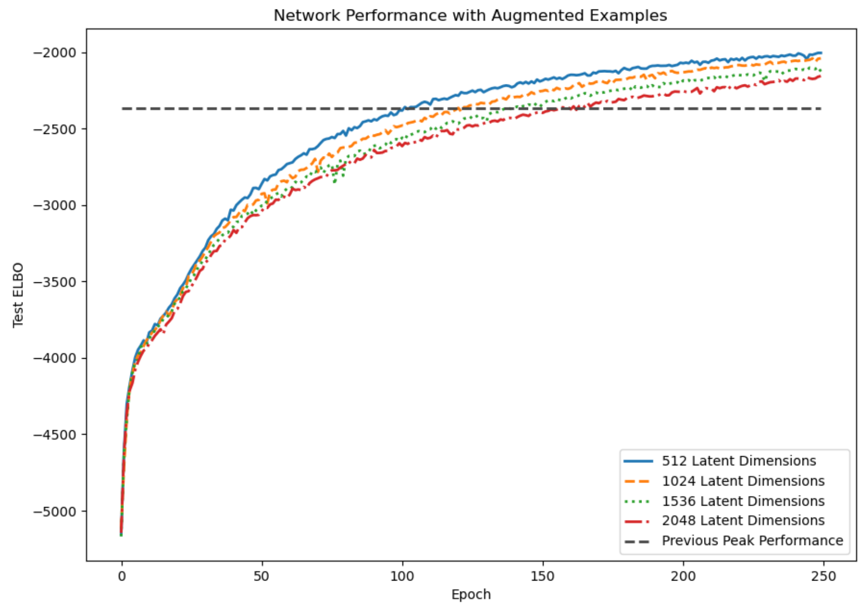
Figure 4.5 A visualization of data augmentation. The original sketch is shown on the left-hand side of the figure, and the augmented sketches are shown on the right-hand side. The top left augmented example is unchanged, the top right augmented example is translated upwards and rightwards, the bottom left augmented example is flipped horizontally, and the bottom right augmented example is rotated.

It should be noted that while data augmentation usually improves the performance of a machine learning algorithm, the augmented training examples may not contribute as much information as would additional training examples taken directly from the set of possible inputs for the learning problem. Moreover, if, after applying the augmentation step, the resulting example is not part of the set of possible inputs, then the augmentation may actually harm the performance of the learning algorithm. I provide simplified illustrations of these two points in Appendix D.

Relating this back to the sketch problem, I decided to only include two augmentations: flipping the images horizontally and translating the images up to roughly a fifth of the image size in both the horizontal and vertical directions. This was because both rotation and scaling augmentations would require interpolation of the pixels of the original image, and I worried that the interpolation distortions might lead to the second of the two problems raised above. I chose to use the highest performing architecture from the non-augmented runs with the same four latent dimensions used in the last section: 512, 1024, 1536, and 2048. Anticipating that these runs would need additional time to fully train, I increased the number of total epochs for training from 150 to 250. The top of Figure 4.6 plots the four test ELBO curves for these training runs.

We can see that all of these networks appear to be approaching the end of their learning, but none of them reach a plateau. We further note that the two runs with fewer latent dimensions appear to be approaching higher quality results more quickly than the two runs with a higher number of latent dimensions. As such, I chose to rerun the networks with 512 and 1024 latent dimensions for 1000 epochs each to find the peak of their performance. To my surprise, both networks once again continued to improve (albeit at a slower and slower rate) through the entire duration of training, as can be seen in the bottom of Figure 4.6. The maximum test ELBO values reached by these two networks were both roughly -1700, and the quality of the test image reprojections was roughly equivalent for both networks. An example test image and its reprojection from the latent space of the augmented 1024 latent-dimension 1000 epoch run are shown in Figure 4.7.

Because the plots of these two runs were nearly identical despite a factor of two difference in the number of latent dimensions, I decided to check the performance of networks with even smaller numbers of latent dimensions. I tried two additional cases, one with the dimensions reduced by another factor of two (256 latent dimensions) and the other with the dimensions strongly reduced to 32. The results of both were unfortunately worse, as summarized in Table 4.4 and shown in Figure 4.8, which indicated that further reduction of the dimensions was more harmful than helpful to the network's performance.



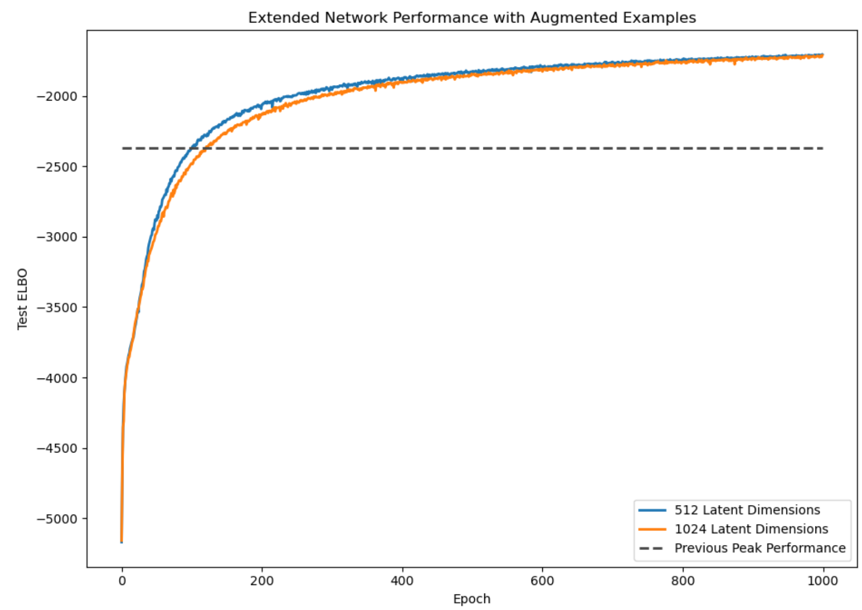


Figure 4.6 (Top) Test ELBO plots for each of the four initial networks trained with data augmentation. These runs significantly outperformed the non-augmented runs; all four surpassed the peak performance from the first set of runs, indicated in the plot by the horizontal dashed line. Even by the end of 250 epochs of training, none of the networks reached a learning plateau. (Bottom) Test EBLO plots for the 1000-epoch runs of the 512 and 1024 latent-dimensional networks trained with data augmentation. Note that even these longer runs failed to reach a learning plateau.

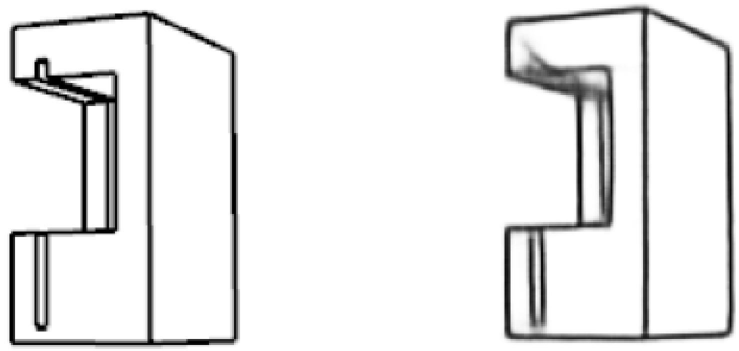


Figure 4.7 A visual comparison of an example test image and its reprojection from the latent space of the 1024 latent dimension, 1000 epoch run using data augmentation. Note that the smaller details now have less blurring than those of the reprojection shown in Figure 4.2.

Augmented Network Performance	
Latent Dimensions	*Maximum Test ELBO*
32	-3149*
256	-1791
512	-1707
1024	-1713

Table 4.4 A performance comparison of the low latent-dimension training runs. *Note that this run was stopped at only 400 epochs, as can be seen in Figure 4.8, whereas the other three were stopped at 1000 epochs.

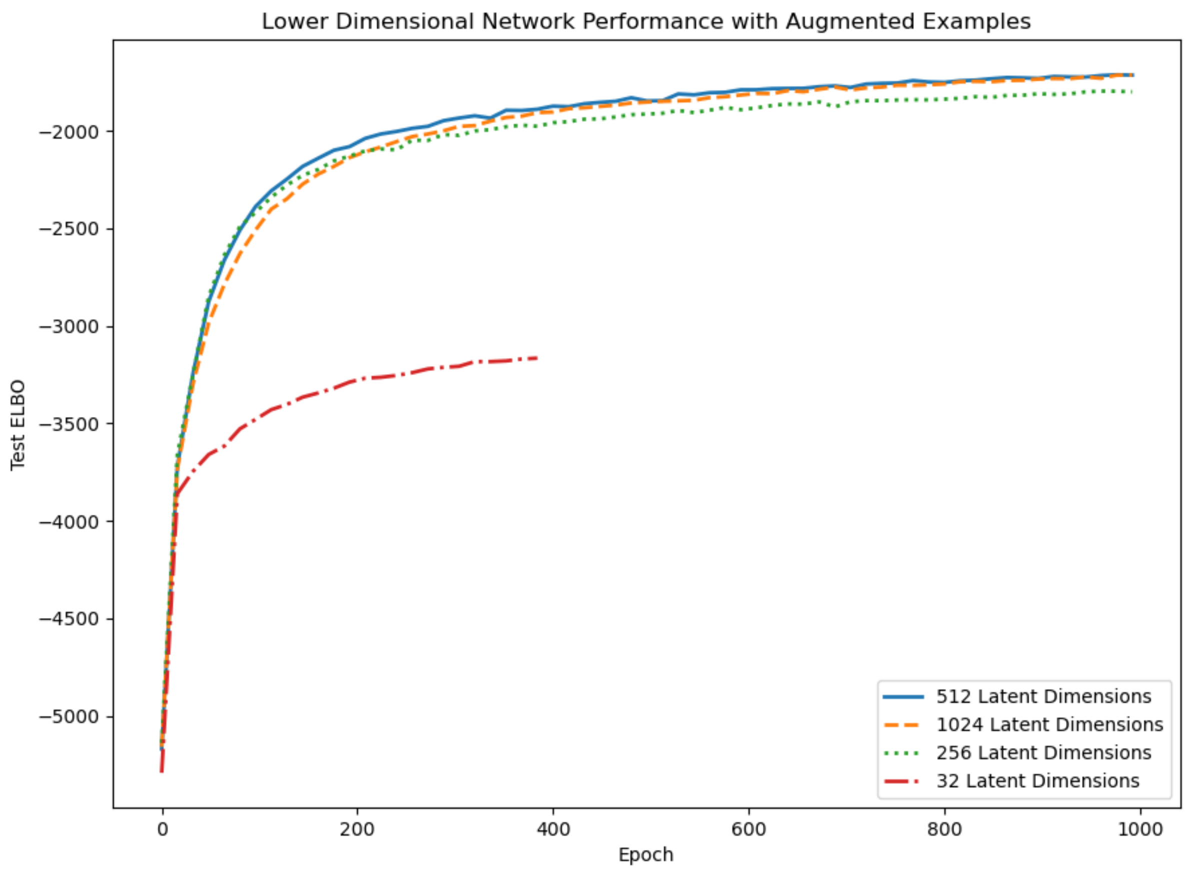


Figure 4.8 Test ELBO plots for the runs with lower numbers of latent dimensions compared to the original two runs with 512 and 1024 dimensions. The 256-dimensional run performed slightly worse than the 512-dimensional and 1024-dimensional runs, and the heavily reduced 32-dimensional run began to level off at a much lower ELBO than the higher-dimensional runs. Note that the plot above was made by subsampling the raw data to enhance visual clarity; the true curves have a significant amount of noise as can be seen in the bottom of Figure 4.6.

As a result of all the data augmentation runs described above, I chose to finalize the four convolutional layer, 1024 latent dimension architecture for generating the latent code representations of input sketches. I chose 1024 latent dimensions over 512 dimensions despite its slightly better performance for two reasons. First, the PCA reprojections for 512 principal components were significantly worse than the 1024 principal components, and given the minor difference in the test performance, it seemed reasonable to use PCA performance as the tiebreaker. Second, the class of models generated by the Grasshopper script is very specific (Boolean operations applied to three axis-aligned rectangular prisms), and a higher latent space dimensionality would theoretically be able to represent a larger set of objects than a smaller latent space dimensionality. Thus, if I chose to broaden the types of models generated in the future, I would expect the higher-dimensional network architecture to be more robust to the change than the lower-dimensional network architecture.

With the latent space encoder in hand, we are now ready to move on to object reconstruction. In the next chapter, we use the latent space representations developed in this chapter as input to a neural network which generates a signed distance field of the object. This SDF can be used to render the 3D object, as discussed in Chapter 6, or to create one of the more standard object representations (e.g., a voxel grid or a mesh).

Chapter 5

Reconstruction from Signed Distance

The past two chapters investigated how to convert sketches to lower-dimensional latent space representations. In this chapter, we will discuss the process of training a signed distance generation network based on the architecture presented in [Park19] that relies on these latent representations. We will train the network on two types of datasets, a smaller, curated dataset that was manually inspected to avoid “bad examples,” and a larger dataset with no manual inspection.

5.1 Curated Dataset Generation

The 3D models used for training the SDF network were generated independently from those used to train the variational autoencoder, and this was done for two reasons. First, it ensured that the performance of the second network would not be influenced by overfitting from the first network. Second, the newly generated examples could be manually inspected/rotated to ensure that they could be unambiguously interpreted by a human, a process which was not necessary for the variational autoencoder datasets.

To elaborate on what makes a sketch "ambiguous" or "unambiguous," consider the examples presented in Figure 5.1. These kinds of ambiguities are common in the datasets that were generated for latent space learning, and so I felt it necessary to manually inspect the entries of the signed distance field dataset to ensure that they were all unambiguous. While one might try to define some specific rules to describe ambiguity and construct an algorithm to perform this inspection automatically, the ambiguity of an example relies on assumptions made by the human visual system which cannot be easily enumerated. Thus, a programmer aspiring to create such an algorithm will run into the same issues raised in Chapter 2 for the overall problem of converting sketches to models.

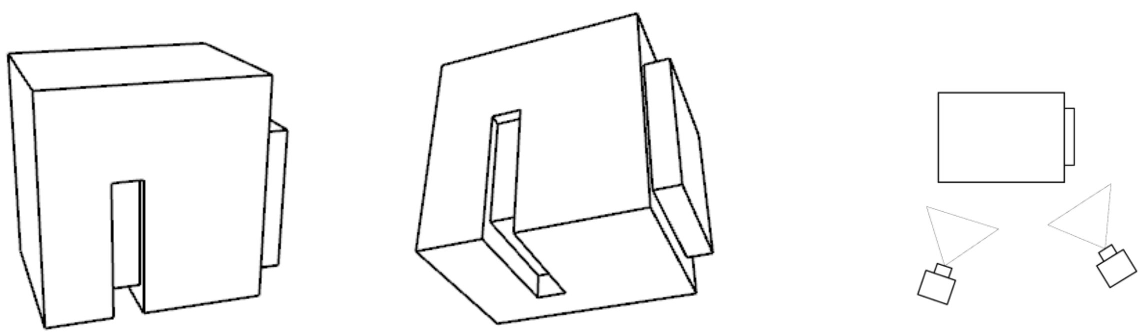


Figure 5.1 Examples of "ambiguous" and "unambiguous" views of the same object. (Left) An ambiguous view of the object, as the depth of the central indent and the width of the extrusion on the right-hand side are unclear from the sketch. (Middle) An unambiguous view of the object, as the extents of the extrusion and indent can clearly be seen. (Right) A top-down view of the two camera viewpoints as a visual aid. Note that the 3D rotations of the cameras are not shown, only their rough 2D top-down rotations are depicted.

Returning to the discussion of dataset generation, the SDF network needed to take in a 1024-dimensional latent vector and an XYZ position as input and return as an output the signed distance to the object. Thus, the dataset needed to have 1027-dimensional features and one-dimensional labels, as shown in Figure 5.2. The following steps were used to generate examples for the dataset: 1) a model was generated exactly as was done for the datasets used in Chapters 3 and 4 with Grasshopper, 2) the 3D space within the model and the 3D space surrounding the model were sampled to generate pairs of points and their signed distances, 3) the model was rendered as a ‘sketch’ and the point-distance pairs were saved to a CSV, 4) the variational autoencoder from Chapter 4 was used to convert the rendered sketch to a 1024-dimensional latent vector.

The sample point generation was done in three steps. First, a 16x16x16 grid of points was sampled around the object, with the coordinate system based on the center of the object as well as relative position and orientation of the camera. Second, 1000 samples of the object’s surface were taken with small deviations so that they were not perfectly on the surface. Finally, 1000 random samples around or within the object were taken. This resulted in a total of 6096 points being sampled per model, all of which were contained within a cube surrounding the object. 200 models were generated and manually inspected, and so the total number of dataset entries was 1,219,200. Visualizations of the point sample generations are shown in Figure 5.3.

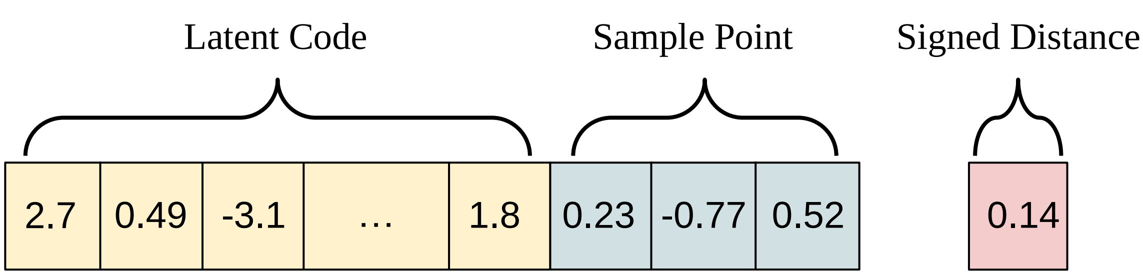


Figure 5.2 An example-label pair of the original dataset used for training. The first 1024 elements of an example are the latent code output of the encoder portion of the trained variational autoencoder. The last three elements are the coordinates of the point being sampled. The label is the signed distance between the sample point and the model surface.

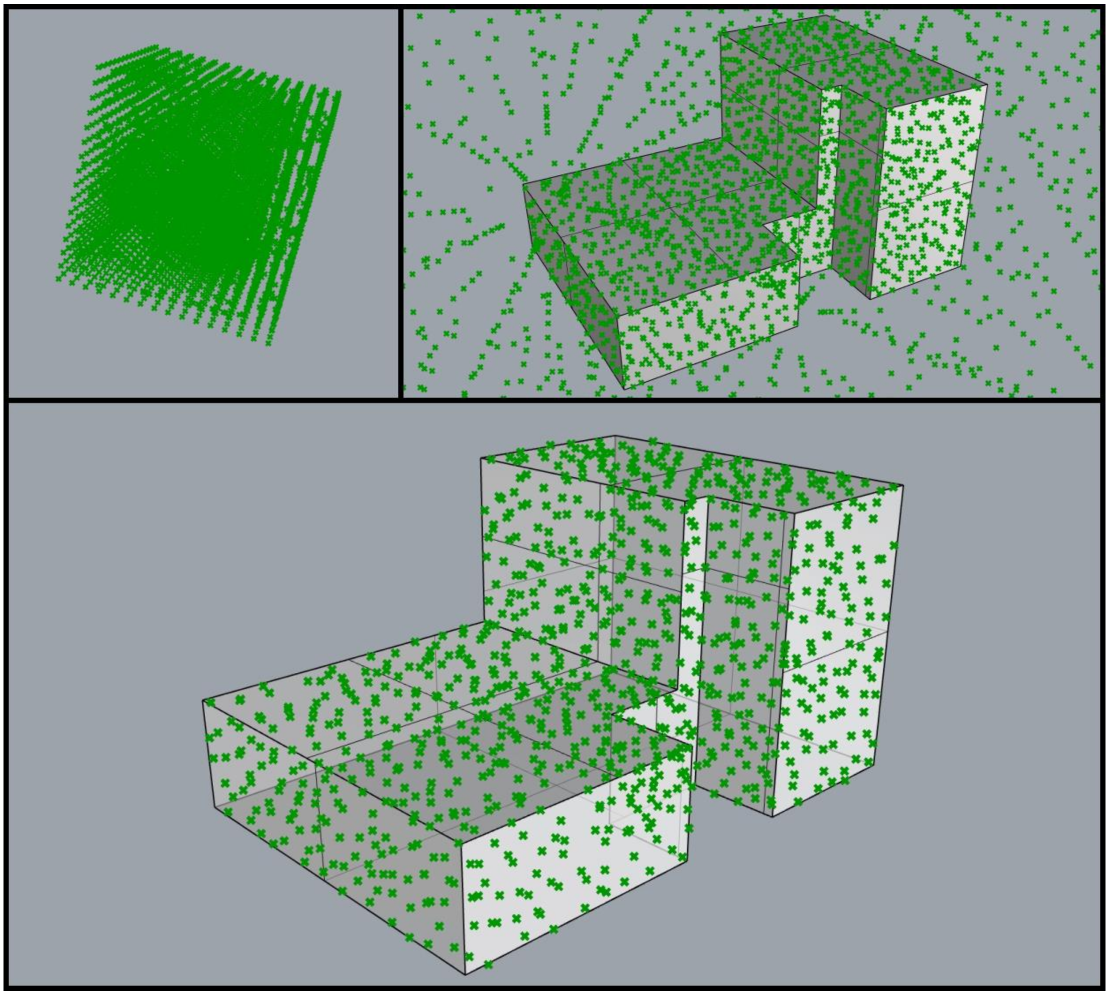


Figure 5.3 Visualization of the point sample generation for a model. (Top Left) The point samples are all contained within a cube with bounds of [-1, 1] in all three directions of the camera-oriented coordinate system. (Top Right) A zoomed in view of the point samples, which can be seen to surround the model; note the higher concentration of points near/on the model. (Bottom) All points outside the model have been removed to demonstrate that the model interior is also being sampled.

5.2 Network Description and Training

Given the successful reconstructions demonstrated in [Park19], I chose a similar network architecture for learning signed distance field representations. A diagram of the original architecture presented in [Park19] is shown at the top of Figure 5.4; it is an eight-layer, fully connected network with a single skip connection that passes the input to the fifth layer. This skip connection allows for the original data to influence the deeper layers of the network more directly.

A diagram of the modified network architecture used in this thesis is shown at the bottom of Figure 5.4. There were two major adjustments I made to the network presented in [Park19]; the first was to increase the size of two of the layers due to the increased length of the latent codes, and the second was to remove the hyperbolic tangent activation layer of the original network. The first change was made to avoid compressing the additional information found in the latent code at too shallow a location in the network, and the second change was made because the cube-like domain of my training examples allowed for signed distances outside the range of -1 to 1, and these would not be possible to recreate with a hyperbolic tangent activation. Instead, I incorporate this hyperbolic tangent into the loss function used for training without making it a layer of the network.

For training the network, the examples were split such that 90% of the models were used for training and 10% were used for testing. As alluded to in the previous paragraph, the runs which achieved reasonable success on the training examples used a hyperbolic tangent loss function. To be more specific, the network's loss for a predicted signed distance 'y_p' and true signed distance 'y' is defined by the following equation:

$$\text{Loss}(y_p, y) = (\tanh(y_p) - \tanh(y))^2$$

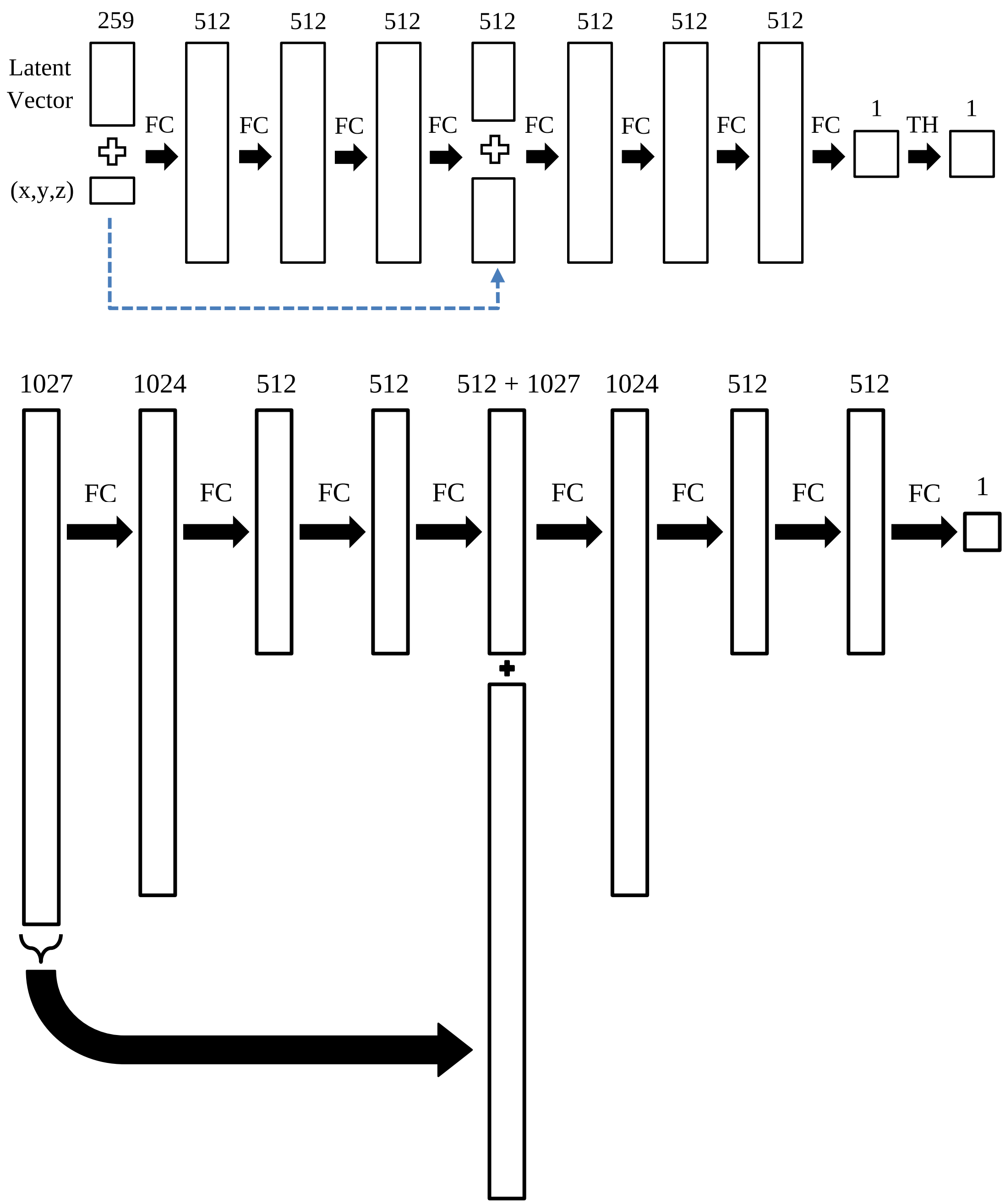


Figure 5.4 Neural Network Architecture Diagrams. (Top) The original network architecture diagram from [Park19]. (Bottom) The modified network architecture used for this thesis work. The major differences between the networks are the output sizes of the first, fourth, and fifth layers as well as the removal of the final hyperbolic tangent activation. In both architectures, each fully connected layer uses dropouts to avoid overfitting, and in my architecture, the dropout rate, or the rate at which neuron activations are randomly switched to zero, is set to 10%.

The reason for including the hyperbolic tangent in the loss function is to help prioritize the accuracy of the signed distances near zero over the raw difference between the reported distances. This is desirable because points on or near the object's surface have near zero signed distances, and so we can expect the region near the surface to have the highest accuracy. The rendering algorithm discussed in Chapter 6 depends heavily on having an accurate surface region, and so this emphasis on near-zero accuracy will help in the final step of rendering.

An ADAM optimizer was used for learning with the initial learning rate set to 0.0001, and it was found that better quality results were obtained if this learning rate were reduced over the course of training. The batch size varied greatly as I fiddled with the network to improve the results, and the first reasonably successful run had a batch size of 200 points. The model trained for a total of 75 epochs, and a plot of the training/testing losses is shown in Figure 5.5.

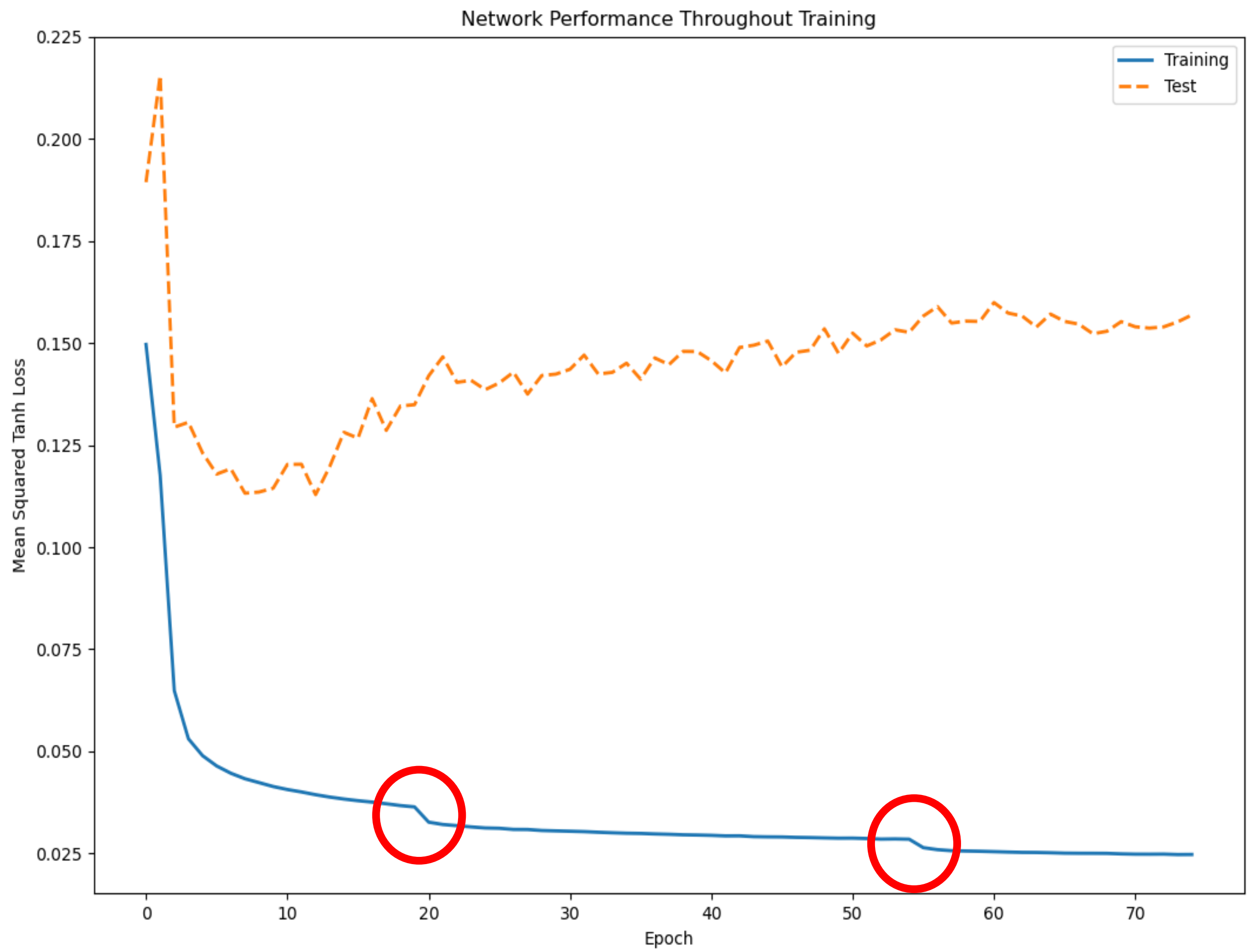


Figure 5.5 Training and test set loss plotted over the course of a training run. Notably, the abrupt changes in loss at epochs 20 and 55 are the result of the ADAM optimizer learning rate being reduced (circled in figure above).

It is clear from Figure 5.5 that the model overfit to the training set, especially considering the loss uses the hyperbolic tangent of the predicted and true signed distances rather than just the raw distances. However, as we've seen throughout this thesis, visual comparisons offer much more information about the quality of the results than the loss plots alone. The specifics of the renderer used to visualize the results will be discussed in the next chapter, but setting aside how the objects were rendered for the moment, Figure 5.6 gives a visual comparison between two sketches from the training set and the ray traced renders of the signed distance fields of the objects. Unfortunately, the signed distance fields generated with sketches from the test set were of such poor quality that the renders yielded either an empty screen or small, disconnected blobs, and as such they were not worth including in a figure.

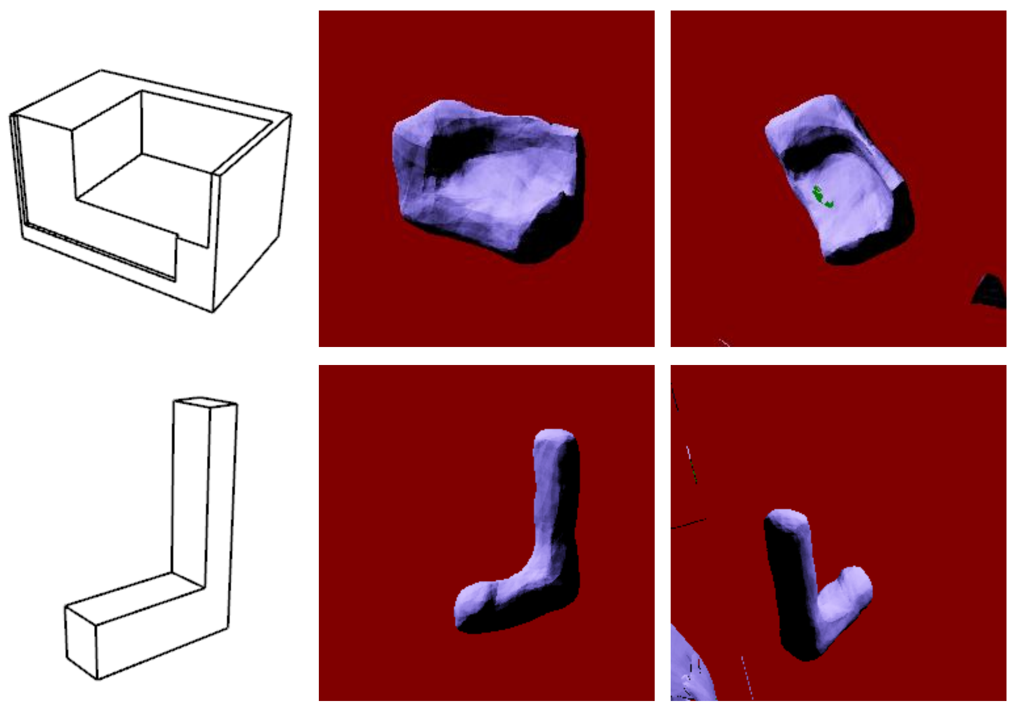


Figure 5.6 A visual comparison of two example training sketches and the rendered signed distance fields. The models are rendered at the original sketch position and at an additional location to show the model's 3D structure. See Chapter 6 for an explanation of how the renderer is implemented. Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

Interestingly, the signed distance field appears to be quite noisy near the surface, as can be seen from the color variation on what should ideally be flat, smooth surfaces, despite efforts to prioritize the accuracy of the surface with the hyperbolic tangent loss. Additionally, for the off-angle renderings, we can see that outside the small cube-shaped domain of the training samples, the network's predictions create all kinds of strange and interesting artifacts. This was addressed with a modification to the rendering algorithm, and is discussed in further detail in Chapter 6.

Another way to visualize the network's signed distance field output is with a voxel grid. While I did choose to avoid them as an output format of the network due to their large size and limited resolving power, they can be used for a simple, coarse visualization of the signed distance field. This visualization method proved invaluable for assessing network performance before the renderer was finished. A 3D voxel representation of the 'L' example sketch created with PyPlot is shown in Figure 5.7.

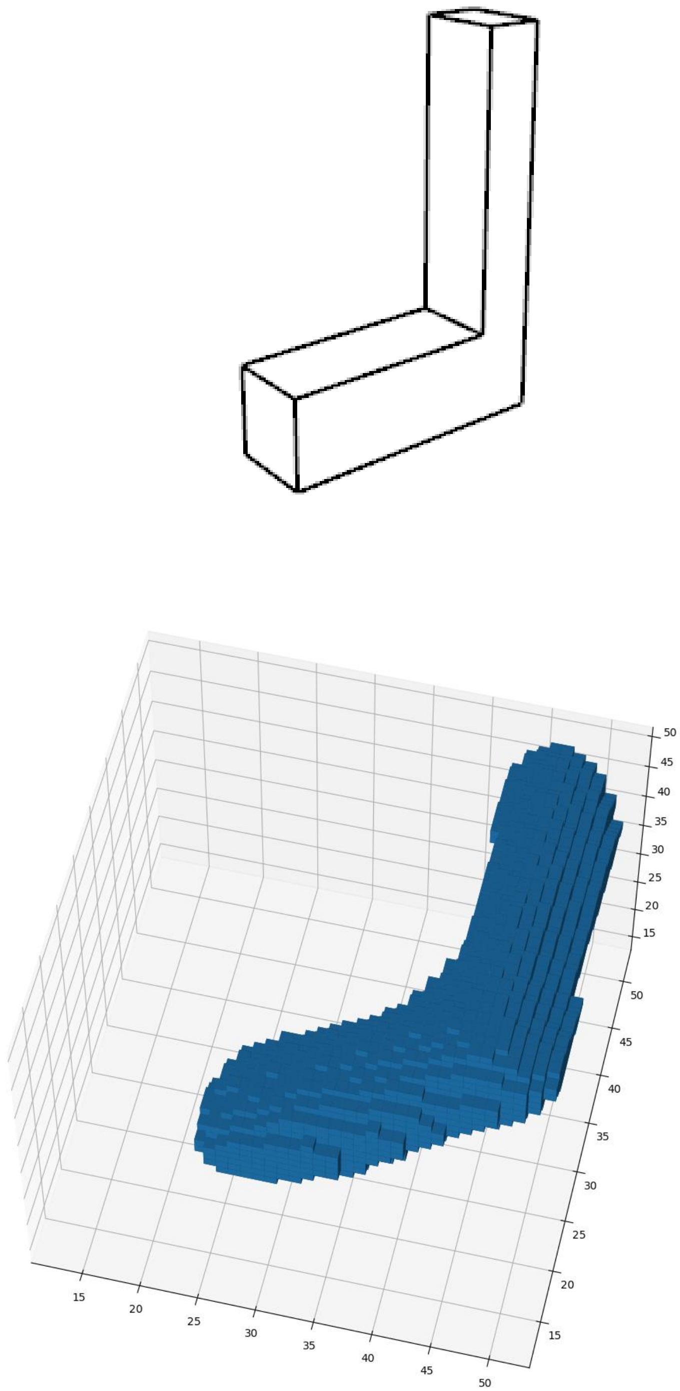


Figure 5.7 Voxelized visualization of the network output for the ‘L’ example. (Top) The ‘L’ sketch used for reconstruction. (Bottom) The voxelized geometry of the reconstructed object. This visualization was achieved by evaluating the output of the network at all grid points and filling in each voxel where the signed distance was less than or equal to zero.

Looking at the visual results in Figure 5.6, we can see that even the training examples significantly underperformed compared to the quality of reconstructions demonstrated in [Park19], which suggests that our network is *underfitting*. However, as stated earlier, the difference in quantitative performance between the training and testing examples suggests the network is *overfitting*. The standard practice for an underfitting network is to *increase* the network complexity and *reduce* the parameters that limit overfitting (for example, the dropout rate), whereas the standard practice for an overfitting network is to *decrease* the network's complexity and to *strengthen* the parameters that limit overfitting.

While we appear to have quite the conundrum on our hands, the contradicting evidence above suggests a few possibilities. First, it is possible that a network of this format cannot learn to solve the sketch-to-model problem, and that the efforts made in the previous chapters were all in vain. As tempting as it would be to throw in the towel, I decided the best course of action was to rule out the other possibilities first. Second, it is possible that the dataset is too small to effectively learn the problem. As such, I chose to significantly increase the size of the dataset, and the remainder of the chapter will discuss the results of training the network on the larger dataset. Finally, it is possible that the latent space developed in the previous chapter is unsuited for reconstruction of 3D objects, as it was only trained on 2D images of sketches. This possibility is not actively investigated in this thesis, but in Chapter 7, I discuss potential ways to address this problem in future work.

5.3 Large Dataset Training

As stated above, one possible reason for the network's performance issues is that the dataset used for training was too small to accurately represent the set of all possible input sketches. However, because the objects/sketches in the original dataset were inspected manually to ensure that they were unambiguous, generating enough additional examples to significantly improve the network's performance would be infeasible.

Thus, I chose to remove the requirement that the sketches be unambiguous and generate a much larger number of examples without any inspection. While this does lead to the generation of "unfair examples" as shown in Figure 5.8, I hoped that, on the whole, the network would learn to generate valid 3D objects that, from the sketch viewpoint, looked approximately the same as the original sketch.

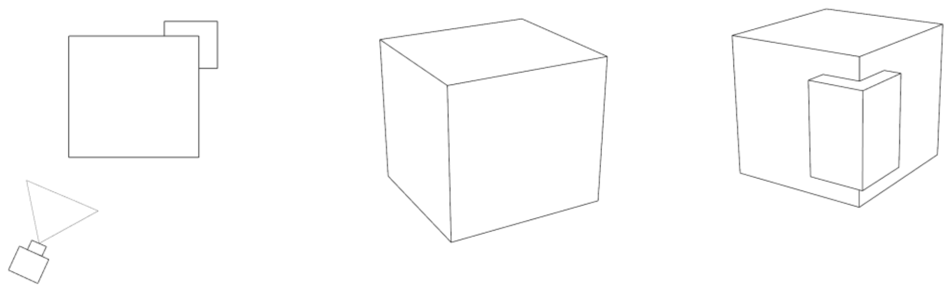


Figure 5.8 An "unfair example" that could be generated by the automatic script if run without manual inspection. (Left) An overhead view of the example, with the camera shown in the bottom left. The occluded geometry on the backside of the model is completely hidden from the camera's viewpoint. (Middle) The sketch that would be generated with the automatic script. (Right) A view of the backside of the model with the occluded geometry.

When generating the larger dataset, the number of surface samples was increased from 1000 to roughly 4000-4500. The exact number of surface points sampled varied from model to model because of a change made to the sample generation algorithm to improve generation speed. The number of grid samples remained unchanged, and additional random samples were taken until the total number of samples was exactly 9000 – a roughly 150% increase in the number of samples per model from the original dataset.

The larger dataset consisted of 10,000 models, and with 9000 sample points each, the total number of entries in the dataset was 90 million. This posed a memory problem; the NVIDIA RTX 3090 GPU on which I had been training my previous networks could not store the entire uncompressed dataset in memory. Thus, I adjusted the storage scheme to a more compressed format; the image codes were stored in a separate, 10,000-entry array, and the 90 million dataset entries were reduced from 1027-dimensions to four – one for the index of the sketch in the 10,000-entry array and three for the XYZ coordinates of the point sample. When training, batches needed to be in the uncompressed format, and so the uncompressed tensor for a given batch was generated from the compressed format and passed as an input to the network. This significantly reduced the memory footprint of the dataset and allowed me to continue training on the RTX 3090 GPU. See Figure 5.9 for a visual representation of the change to the storage scheme.

While most of the training hyperparameters were identical for the larger dataset, the batch size was increased from 250 to 450, and the ADAM learning rate was adjusted dynamically rather than at specified epochs. The change in loss from epoch to epoch was actively monitored, and if the change in loss was less than a hyperparameter threshold twice in a row, the learning rate and threshold were both halved. The threshold hyperparameter was initially set to 0.001, ten times the initial ADAM learning rate of 0.0001. A plot of the training/test losses for a large dataset training run is shown in Figure 5.10, and a comparison to the training/test losses of the original, smaller dataset training run is shown in Figure 5.11.

Sketches and renders of the generated signed distance fields from the default camera position are shown in Figure 5.12, Figure 5.13, and Figure 5.14 for visual assessment of the results. The SDF renders generated from the training sketches remained largely faithful to the overall form depicted in the sketch, though many of the smaller details were completely lost. In the original dataset, the test set results were unable to be rendered due to their poor quality, but because of the huge reduction in test error for the larger dataset, the test results from the larger dataset run were able to be rendered. Because the models of the original smaller dataset were not part of the larger 10,000 model dataset, the sketches from the original dataset could be used as unseen test examples to compare performance between the two runs. The quality of the generated SDFs for test sketches varied significantly, and Figure 5.13 and Figure 5.14 show the range of quality of the reconstructions.

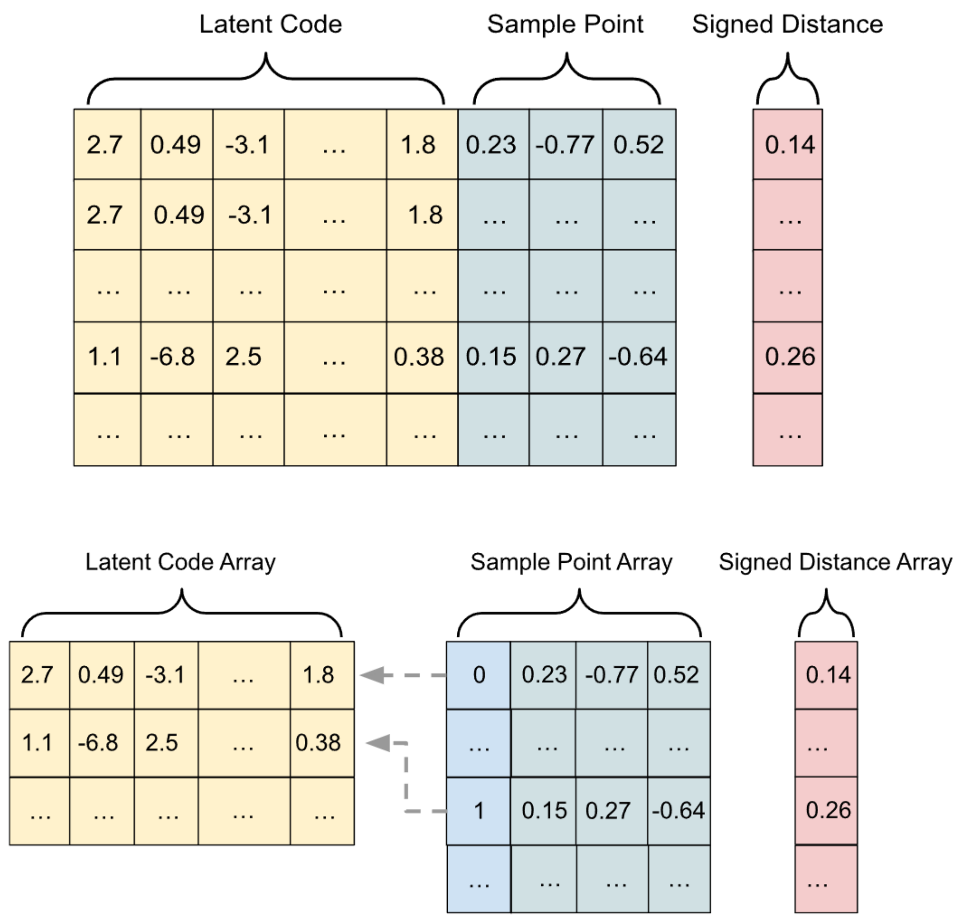


Figure 5.9 Block diagrams of the uncompressed and compressed dataset storage schemes used for SDF generation training. (Top) The original, uncompressed dataset format. Each latent code is repeated for each point sample, which is highly redundant and requires a large amount of memory. Batches must be converted to this format to be passed into the network. (Bottom) The compressed dataset format. Each image code only appears once in the latent code array, and the sample points are augmented with the index of the corresponding latent code in the latent code array. Although they are shown as sorted arrays in the diagrams above, in practice the entries are randomly shuffled before training.

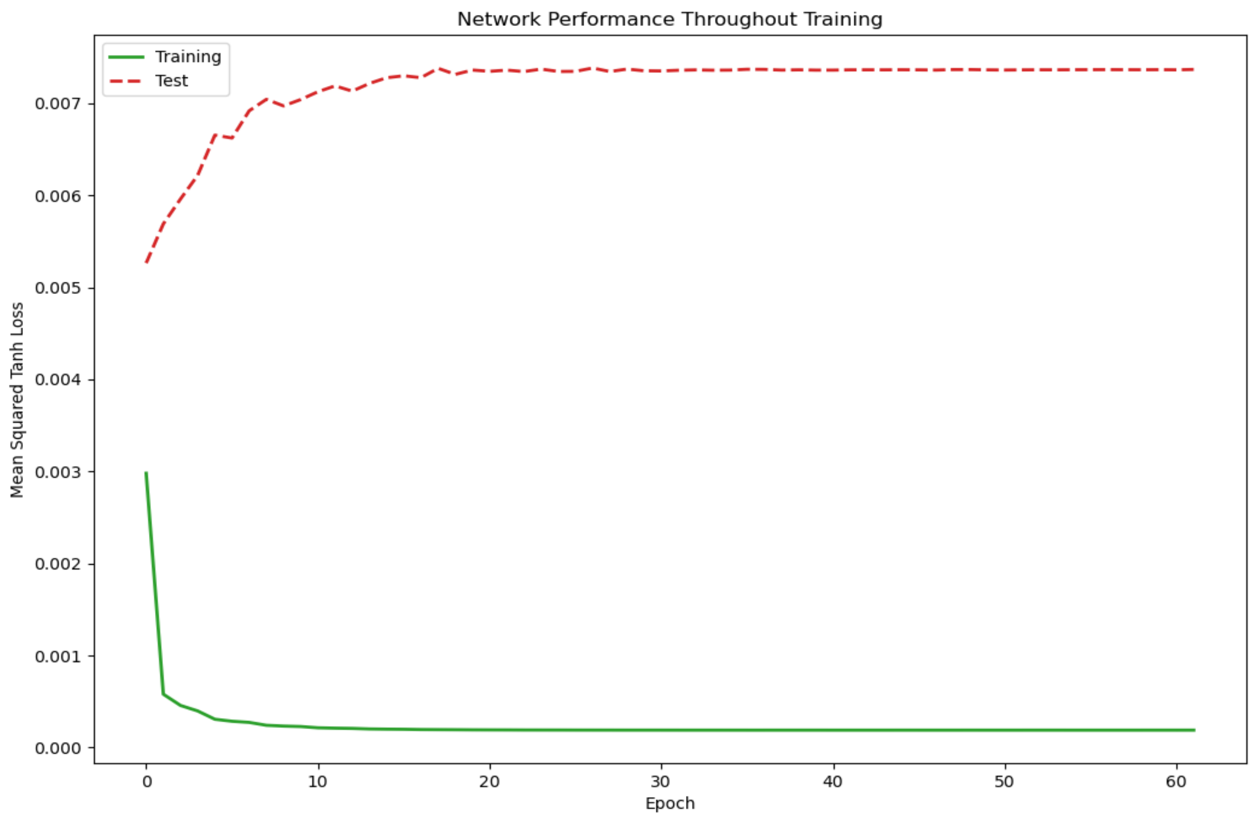


Figure 5.10 A plot of the training/test losses of the network for the large dataset training run. Unfortunately, at the time the data was recorded, the initial loss of the untrained network was not recorded, and so the loss curves start after the first training epoch. Because of the rapid speed of convergence with the larger dataset, the test error decreased to its lowest value within this first epoch, leading to the odd-looking plot of the test error above. While the exact value of the initial loss for the run shown above is unknown, subsequent tests with random initializations yielded initial losses that varied from 0.13 to 0.68.

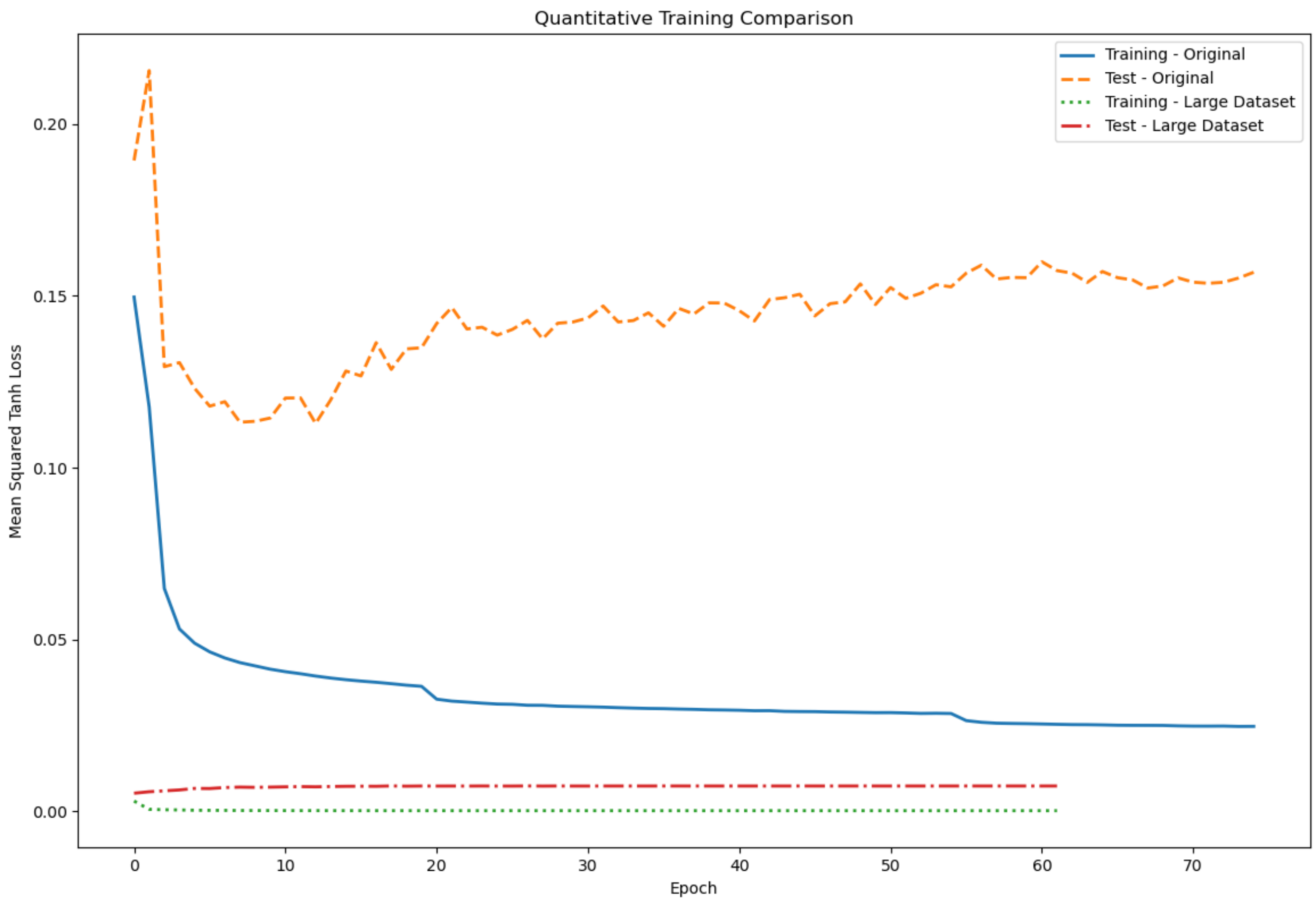


Figure 5.11 Comparison of the training/testing losses between the smaller and larger datasets. Notably, both the training set and test set losses of the larger dataset run are significantly lower than the training set losses of the smaller dataset run. The larger dataset run was stopped earlier than the smaller dataset run as the learning process had reached a plateau.

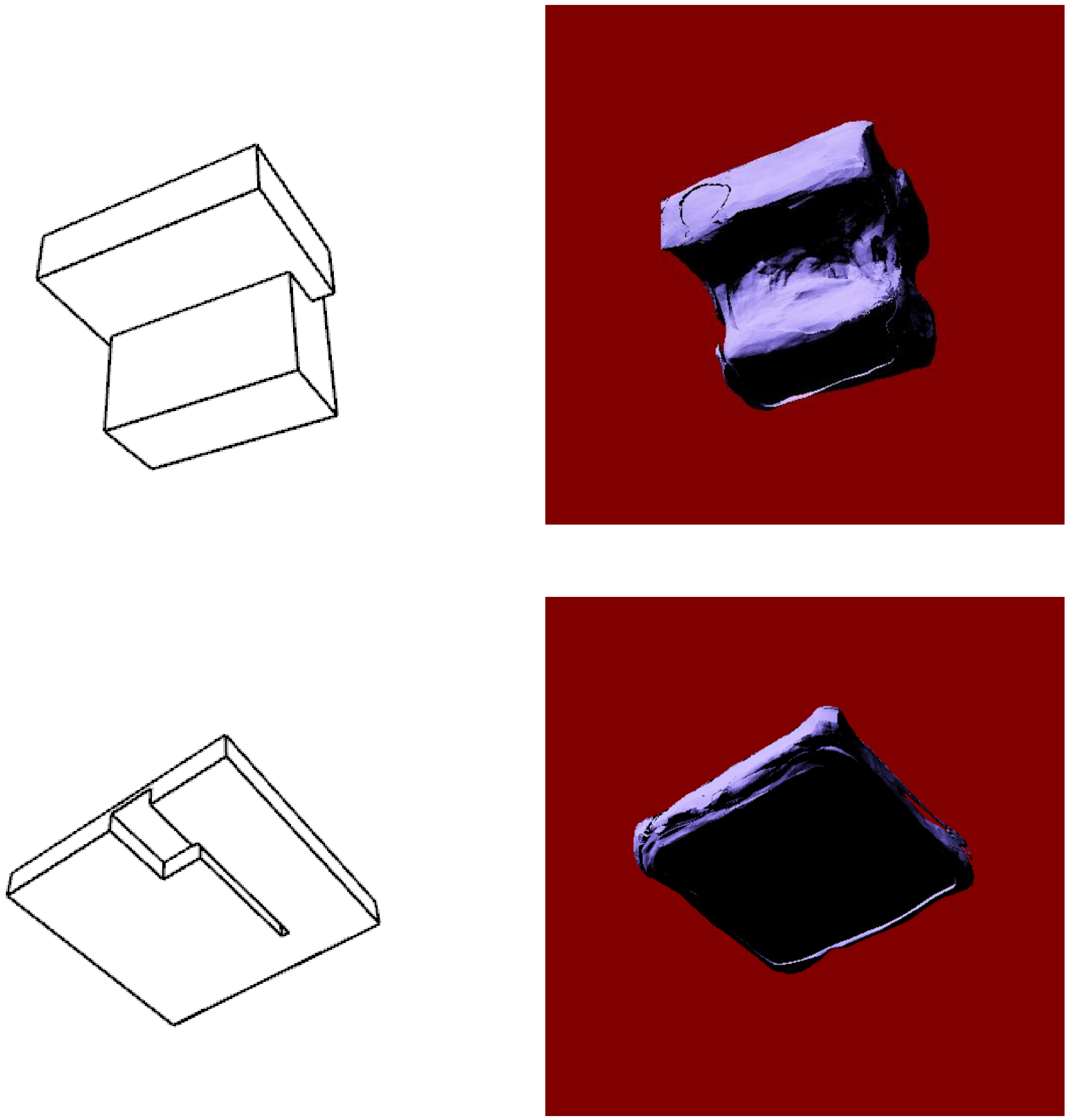


Figure 5.12 Example sketches (Left) and renders of the generated signed distance fields (Right) from the training set of the large dataset run. The generated objects have the correct overall form, but tend to lose the smaller scale details. Interestingly, the lower sketch includes an unintentional optical illusion for some viewers - because of the viewpoint at which the sketch was generated, the left half of the object's base appears to be occluding the right half, even though the base is entirely level. Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

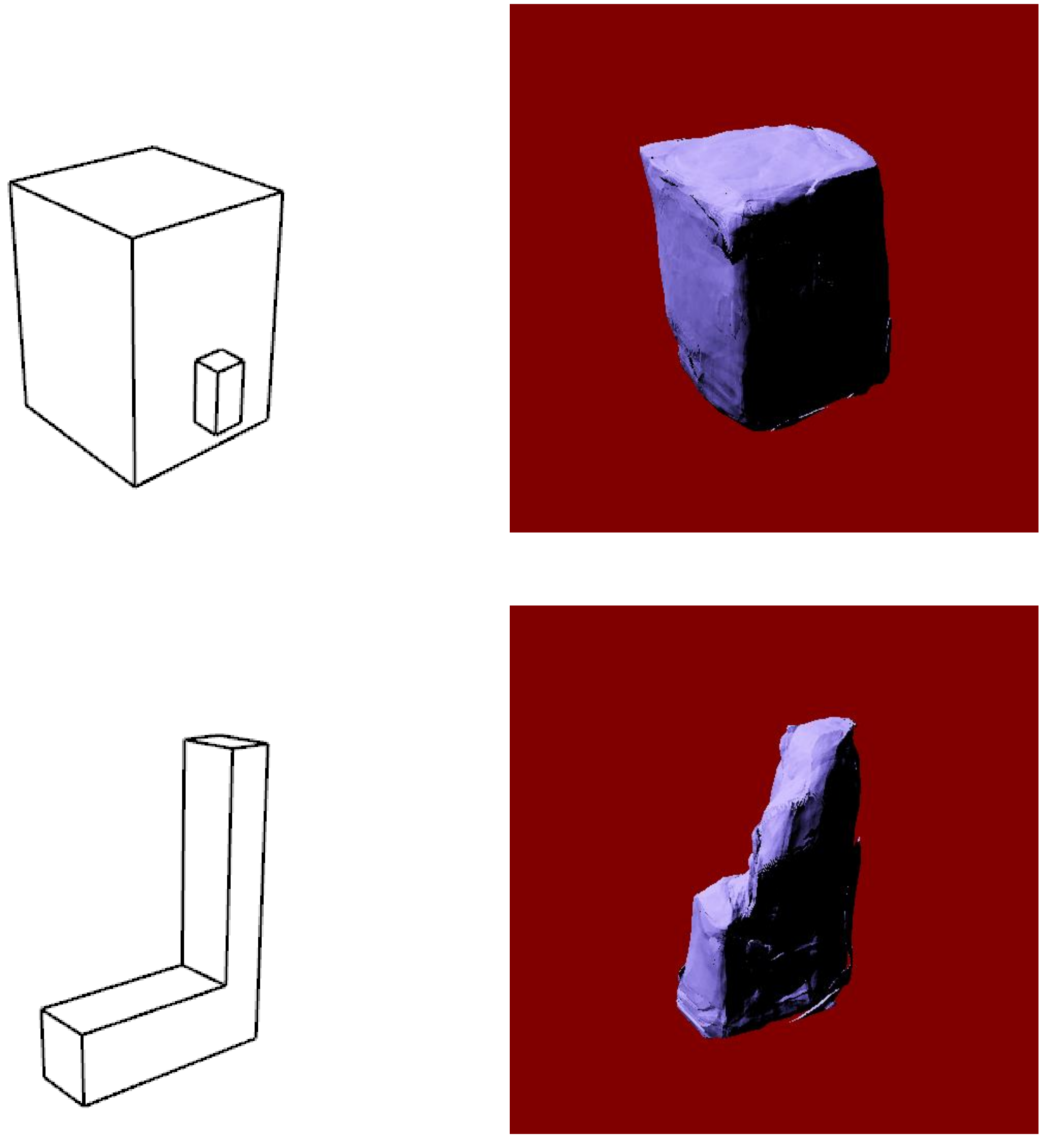


Figure 5.13 Example sketches (Left) and renders of the generated signed distance fields (Right) from the test set. These examples are two of the highest quality test sketch reconstructions inspected, as they maintain general form of the object in the sketch. However, there is clearly significant room for improvement. It is interesting that although the test performance of the larger dataset run was quantitatively better than the training performance of the smaller dataset run (see Figure 5.11), the rendered surface of the ‘L’ sketch is significantly less accurate for this run than it was for the smaller dataset training run (see Figure 5.6). Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

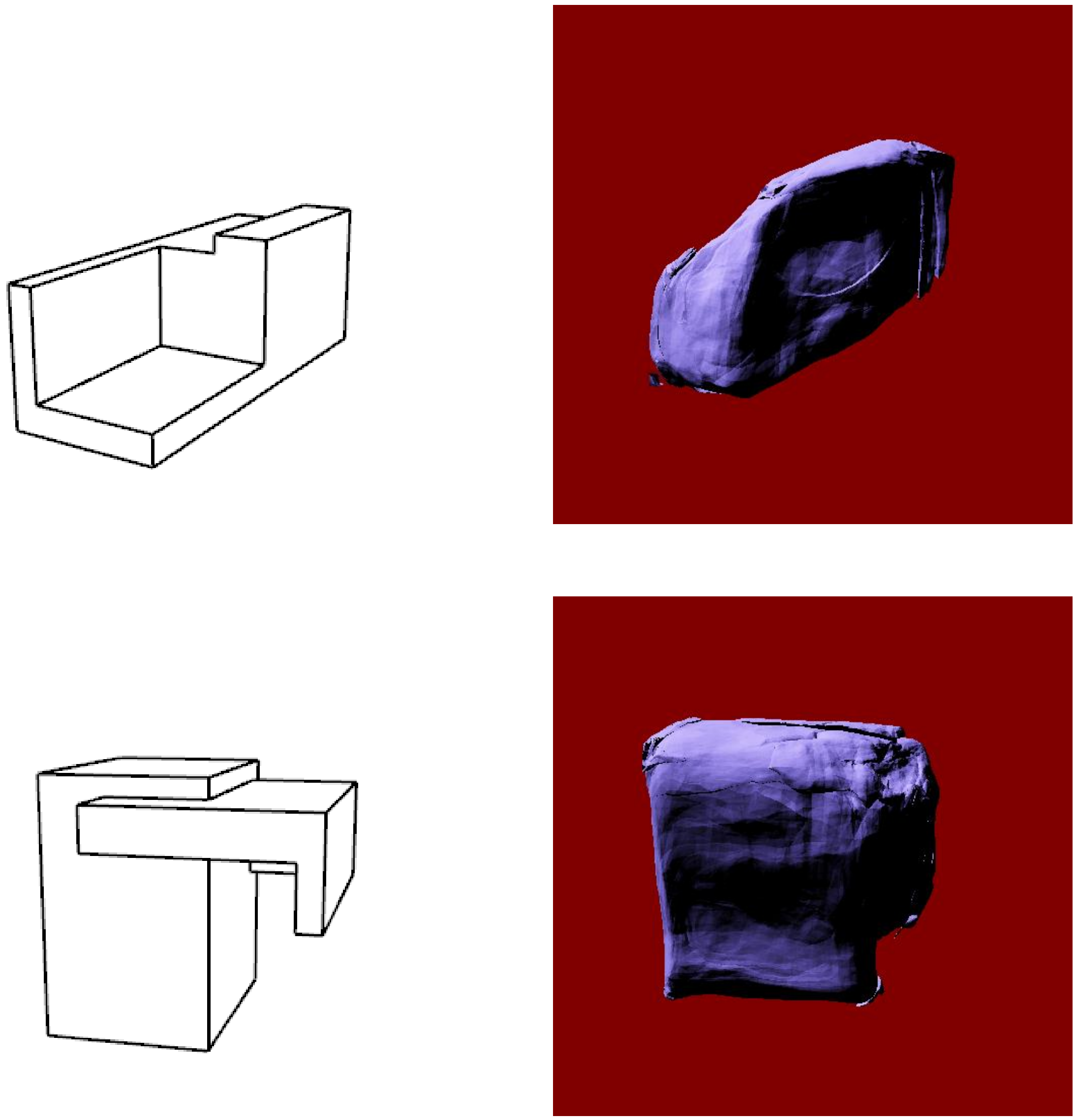


Figure 5.14 Example sketches (Left) and renders of the generated signed distance fields (Right) from the test set. These examples are two of the lowest quality test sketch reconstructions inspected. The SDF renders do appear to match the “convex hull” of the objects depicted in the sketches, but the loss of detail is much worse than most of the other inspected examples. Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

Although training on the larger dataset led to remarkable improvement for unseen sketch reconstruction compared to the original smaller dataset run, the quality of the results still pales in comparison to the quality presented in [Park19]. One likely possibility is that the dataset is still too small. The network presented in [Park19] was trained on models from the ShapeNet dataset, which contains more than three million models, and even though many of these were discarded because they were not watertight, it is reasonable to assume that the number of models used to train the DeepSDF network was 1-2 orders of magnitude higher than the number of models used in this thesis work. Moreover, according to the paper, 500,000 sample points were used for each model compared to the paltry 9000 points used in this thesis. Thus, it *is* possible that, given a dataset of the same scale, the network presented in this work would have comparable results. However, as stated earlier, the quality of the latent space representations is critical to the success of the reconstructions, and so it is also possible that the learned latent space is responsible for the shortcomings of the reconstructions. This possibility is discussed further in Chapter 7.

Once the signed distance field of a sketch has been generated, the last step of the process is to render the surface encoded in the SDF representation. The relatively simple ray-marching algorithm used for rendering and the various modifications made to improve the algorithm's robustness to noisy SDFs are the focus of the next chapter.

Chapter 6

Ray Tracing Implicit Neural Fields

In the previous chapter, we saw lighting-based renders of the objects represented by the generated signed distance fields. In this chapter, I describe the implementation of the ray tracer used to create those renders. The ray tracer relies on a modified form of *sphere tracing*, a standard ray-marching technique used to render well-behaved signed distance fields. Unfortunately, the output of the neural network was often poorly-behaved, and additional custom modifications were necessary to assist the convergence of the rays to the object's surface.

6.1 Ideal Sphere Tracing

Sphere tracing is a simple but powerful tool for rendering implicit surfaces. [Hart95] proposes a modification of the standard ray tracing algorithm for implicit surfaces; instead of finding all intersections of a ray as is done in a standard ray tracer, the ray is "stepped" out to the first surface it intersects. The idea is simple: start at some starting point $t = t_0$ along the ray and iteratively add the signed distance evaluated at the current position along the ray, t_n, to compute the next position along the ray, t_{n+1}, until an intersection is found. The process can be repeated sequentially for each ray or done in parallel for all desired rays. For those unfamiliar with the algorithm, it may not be immediately obvious why this approach is guaranteed to find the first intersection, and so to help address any confusion, I briefly walk through an example in Appendix E.

One of the main issues with sphere tracing is that near misses lead to a large number of iterations with small step sizes. This is primarily a concern for rendering speed, but because an interactive ray tracer is unnecessary for the purposes of this thesis work, I saw no reason to overcomplicate the renderer. I also decided to keep the maximum iterations quite high for the same reason. With this approach, and a few self-derived mathematical formulas for the signed distance field of a cube, I was able to create the render shown in Figure 6.1.

Figure 6.1 The Treachery of Rendering (caption added with photo-editing software). Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

Fortunately, the SDF network from Chapter 5 outputs exactly what sphere tracing needs – the signed distance to the surface of the object – and so the sphere tracing renderer was adjusted to iteratively query the SDF network along rays to determine new ray positions until the surface was found. To speed up computation time, the renderer computes all the rays for every pixel in the image in parallel, stopping for a given pixel when the distance to the surface from the current ray position is within a small user-defined threshold ε. If all pixels reach this threshold or step too far into the distance before the maximum number of iterations, then the ray tracing loop ends early, otherwise it continues for 1000 iterations. Once the iterative loop terminates, if the signed distance evaluated at the ray position is within the threshold ε, the pixel is colored based on a simple Lambertian shading model with a single hard-coded directional light source. The normal of the surface is computed as the gradient of the signed distance field with respect to the 3 spatial coordinates, which can be computed from the network via back-propagation. For rays that step beyond the bounds of the model, the pixel is colored with a background color, and for rays that fail to reach a surface or the background within 1000 iterations, the pixel is colored with an error color. The background, error, and object colors are chosen arbitrarily and can easily be switched.

6.2 Specialized Modifications

For well-behaved signed distance fields, the above method should work with very little difficulty. Unfortunately, in practice the outputs of the signed distance generation were rarely well-behaved, and the algorithm needed additional modifications to handle some of the issues encountered. The two main issues addressed with these modifications were the non-convergence of patches of rays and extra-domain artifacts.

An example of a patch of non-converging rays rendered with the unmodified setup is shown in Figure 6.2. Looking at the plot of the ray position over time of the problematic pixels shown on the lefthand side of Figure 6.3, the network appears to overestimate the signed distance to the surface, and so the ray position bounces back and forth across the surface in a never-ending cycle instead of approaching it. The first attempt to resolve this problem identified repeated ray positions within a small tolerance and placed the ray position at the halfway point between the two repeating positions, but this led to the behavior shown on the righthand side of Figure 6.3. Please see the figure’s caption for more information.

In light of this observation, two additional modifications were made. First, to avoid the oscillating behavior occurring on a large scale, the step size was clamped to a maximum allowable magnitude. Second, to help avoid problematic large bounces after a bisection, the step size was scaled down by a factor of four after a bisection to counteract this effect. Together, these two changes eliminated the vast majority of the non-converging rays, as can be seen in Figure 6.4.

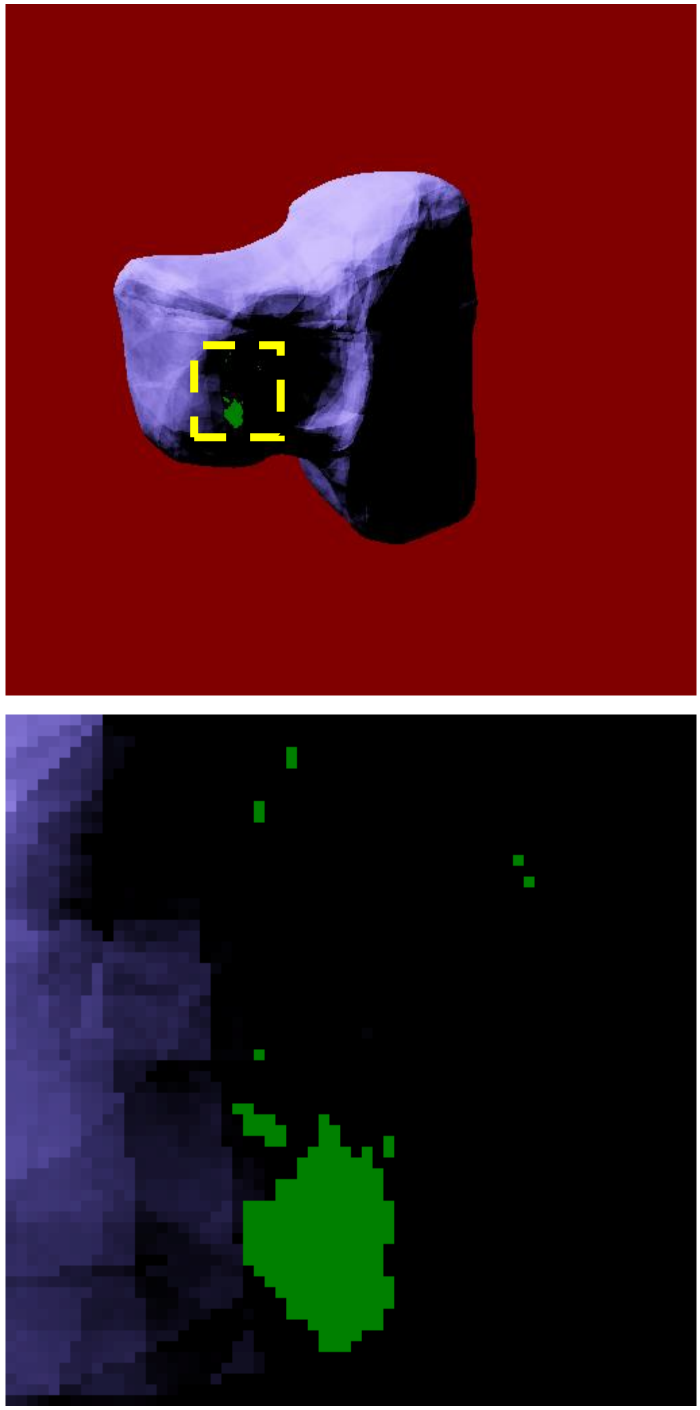


Figure 6.2 (Top) An example rendering with non-converged pixels (outlined in yellow). The pixels colored green indicate that sphere-tracing failed to converge. (Bottom) A zoomed-in view of the problematic area. Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

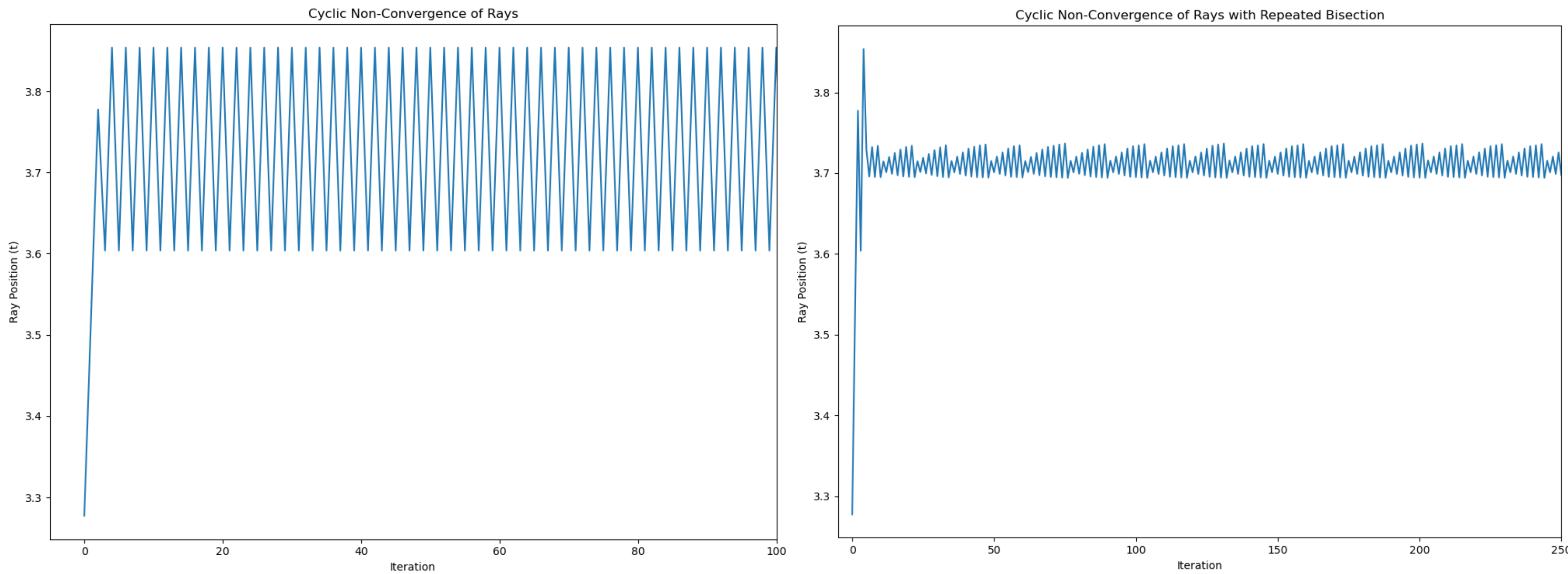


Figure 6.3 (Left) A ray position plot for a non-converging ray with basic sphere tracing. The ray's position oscillates between two values until the maximum number of iterations. (Right) A ray position plot for a non-converging ray with naïve midpoint ray correction. The bisection does put the ray position closer to the surface, but because the network's output signed distance is still large after the bisection, the ray position continues to oscillate across the surface, growing further away with each step until it reaches the original equilibrium. At this point, the repeated positions are bisected again, the ray position bounces back up to the equilibrium again, and the pattern repeats until the maximum number of iterations. Note that only the earlier iterations are shown on the plots above to improve visual clarity.

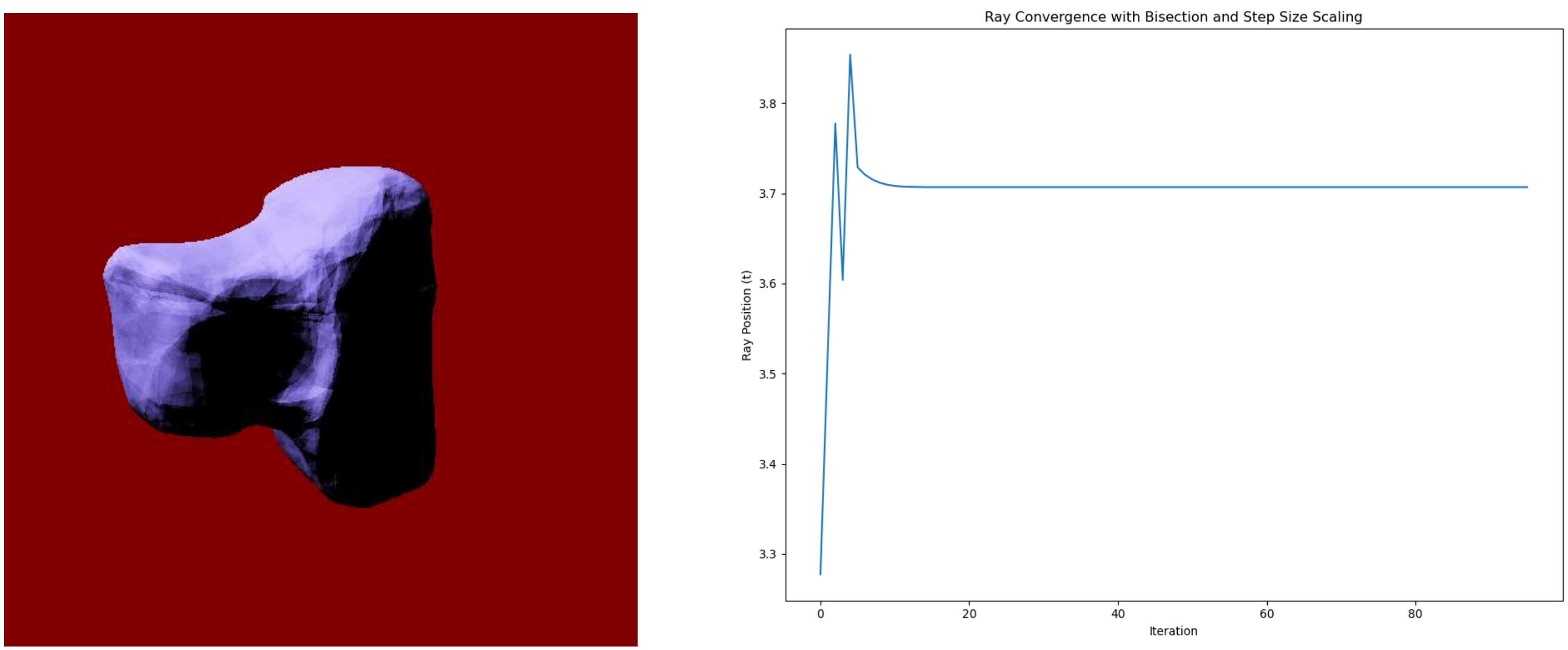


Figure 6.4 (Left) Rendering with the convergence-assisting modifications. All green non-convergent pixels have been eliminated. Note that the colors in the figure have been brightened to improve the quality of printed copies of this work. (Right) A ray position plot for the formerly non-converging ray, rendered with the additional modifications. Instead of permanently oscillating, the ray position approaches the surface after crossing it three times.

It is worth noting that one of the main reasons for the initial non-converging behavior was human error in data preparation rather than a failure of the training process. Due to an oversight in the data generation script, the reported signed distances of the original, smaller dataset were incorrectly scaled and were often much greater in magnitude than the true signed distances. This was fixed for the larger dataset, but because the bisection modification did not appear to be negatively influencing the quality of the renders, it was kept for the larger dataset renders as well.

The second issue, extra-domain artifacting, was the result of the limited domain on which the network was trained. Because all the training examples were sampled within a uniform cube, the behavior of a generated signed distance field outside this cube is uncontrolled and full of artifacts. While not particularly useful in a reconstruction sense, the extra-domain shapes do make for fascinating visuals, as shown at the top of Figure 6.5. Fortunately, removing these artifacts is a simple matter of checking whether a ray intersection lies within the machine learning cube domain, and rendering the pixel with the background color if it does not. The bottom of Figure 6.5 shows the same reconstruction with the artifacts removed.

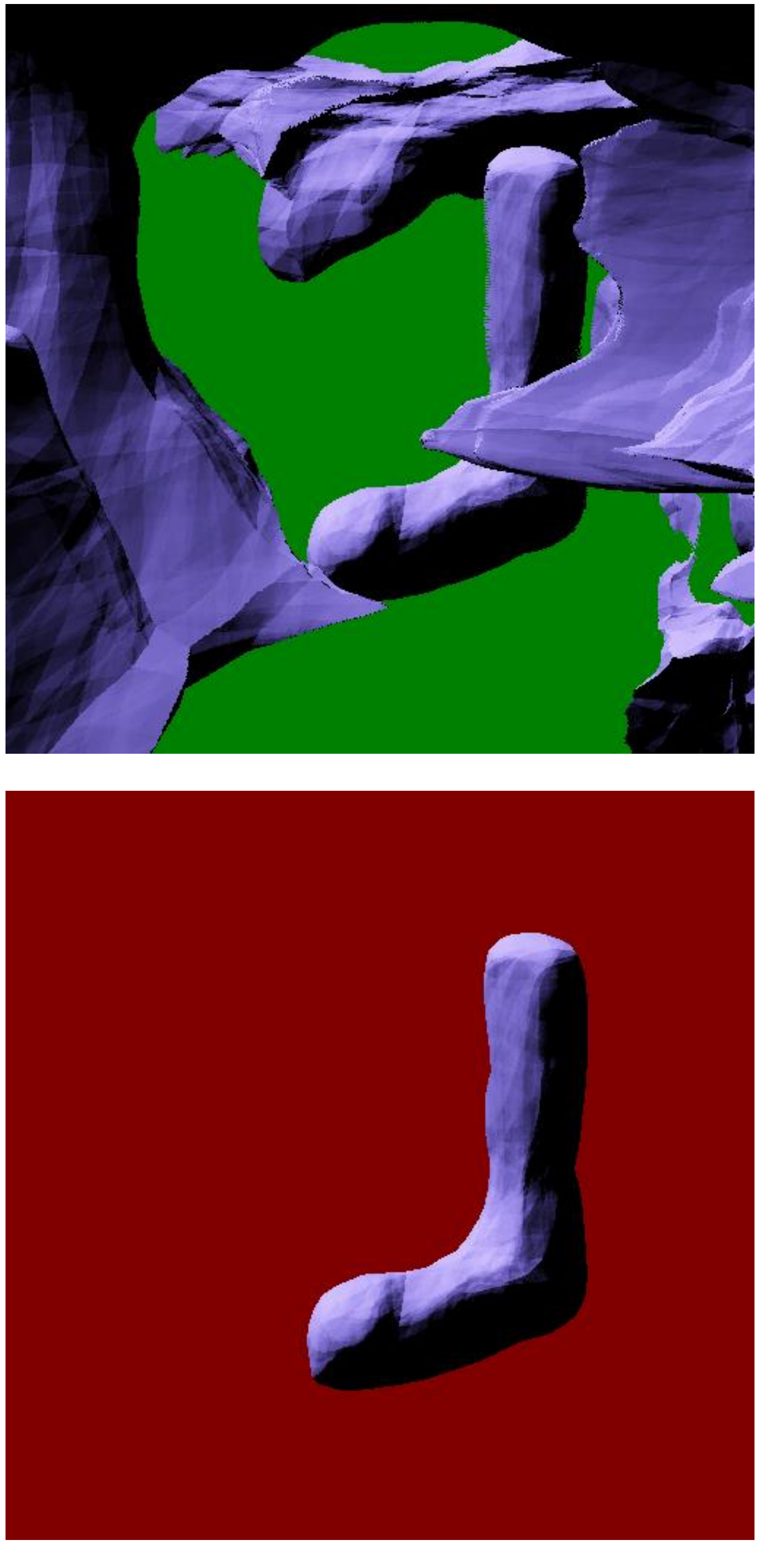


Figure 6.5 A rendering of the ‘L’ example with the extra-domain filtering turned off (Top) and turned on (Bottom). Note that the colors in the figure have been brightened to improve the quality of printed copies of this work.

Chapter 7

Current Shortcomings & Future Work

Although the sketch-to-model process described in the previous chapters performs well considering its many simplifying assumptions, the quality of the results is still too poor to be useful as a software tool. In this chapter, I discuss possible explanations for the limited capabilities of the neural networks described in the previous chapters and how one might resolve these limitations in future work. I start with a discussion of the limitations of the learned latent space and propose a high-level framework for an improved latent space for the sketch-to-model problem. Then, I discuss additional shortcomings of the process and possible ways to address them.

7.1 Latent Space Limitations

In the opening paragraph of Chapter 3, I stated that a latent space learned purely on 3D models would be problematic. The reason for this is because a 2D line-drawing can represent any number of vastly different objects that appear identical from a particular viewpoint, and a purely model-based latent space would fail to cluster all the possible objects for a given sketch into a localized region of the latent space. Because the possible objects for a sketch aren't clustered together, an encoder would have great difficulty identifying all the possibilities for a particular sketch. Please see Appendix F for a more in-depth discussion of this reasoning.

Despite having a strong argument for using a sketch-based rather than model-based latent space, the sketch-based latent space developed in Chapter 4 is problematic in its own ways. In Chapter 3, one of the arguments I made for the use of the learned latent space was to create a representation that held more semantic meaning than that of the spaces defined by principal components. Specifically, I noted that the norm of the difference in the latent representations of the two cubes in Figure 3.8 was nearly twice as large as the norm of the difference between the middle cube and the irregular polyhedron. Then, in Chapter 4, I claimed that the convolutional network would, in theory, create latent spaces with some degree of shift-equivariance.

Unfortunately, this line of reasoning doesn't hold up when put to the test. Comparing the learned latent space reprojections of the three sketches in Figure 3.8, we see that the two cubes still have a larger difference norm than that difference norm of the middle cube and the irregular polyhedron. The three representations' difference norms are shown in Table 7.1. This indicates that the sketch latent space formed by the VAE still fails to recognize the similarity between the left and middle sketches, and thus the latent representations still lack the "semantic meaning" I had hoped to incorporate by using the convolutional learning-based approach.

Another issue with the latent space is its poor interpolation behavior compared to the latent spaces of the literature reviewed in Chapter 2. [Ha17] and [Wu16] demonstrate the quality of their latent spaces by generating the encodings of two inputs, interpolating between these input encodings, and decoding the resulting latent vectors. These decoded interpolations demonstrate the smoothness of the latent spaces they present. Unfortunately, running a similar test on the latent space developed in Chapter 4 yields less than impressive results as can be seen in Figure 7.1. The sketches fail to morph as is seen in the surveyed literature; instead, one sketch simply fades out while another fades in.

Sketches Compared	**Pixel Difference Norm**	**Eigensketch Difference Norm**	**Learned Latent Space Difference Norm**
Left & Middle	38.376	23.273	22.556
Left & Right	41.355	22.895	25.979
Middle & Right	26.784	10.232	18.973

Table 7.1 An augmented summary of the magnitudes of the feature differences between each possible pair of images from Figure 3.8. The first two columns are repeated from Table 3.1, and the third column lists the difference norms for the sketches' learned latent representations using the VAE described in Chapter 4. The ratio of the norm difference of the left and middle sketches to the norm difference of the middle and right sketches is 1.19, which is smaller than 1.43, the value of the ratio of the raw images' difference norms, but it is still greater than one. Thus, the learned latent space stills considers the two cubes to be more different than the middle cube and the irregular polyhedron.

Furthermore, there are plenty of sketch examples outside the training/test datasets used for training the VAE that have poor reprojections from their latent space encodings. Because the training examples were drawn at the center of the image, reprojections for off-center sketches often lose information about the edges of the sketch. This effect was reduced by the translation-based data augmentations described in Chapter 4, but loss of sketch details still occurred on some examples like the one shown at the top of Figure 7.2. Moreover, because all the sketches used for training/testing had identical line thicknesses, the reprojections of sketches that use a different line thickness become distorted, as shown in the bottom of Figure 7.2. These are just two of the many failure modes of the learned encodings, and these examples of failures suggest that the network's latent representation lacks important information that would likely aid in reconstructing the geometry of the represented objects.

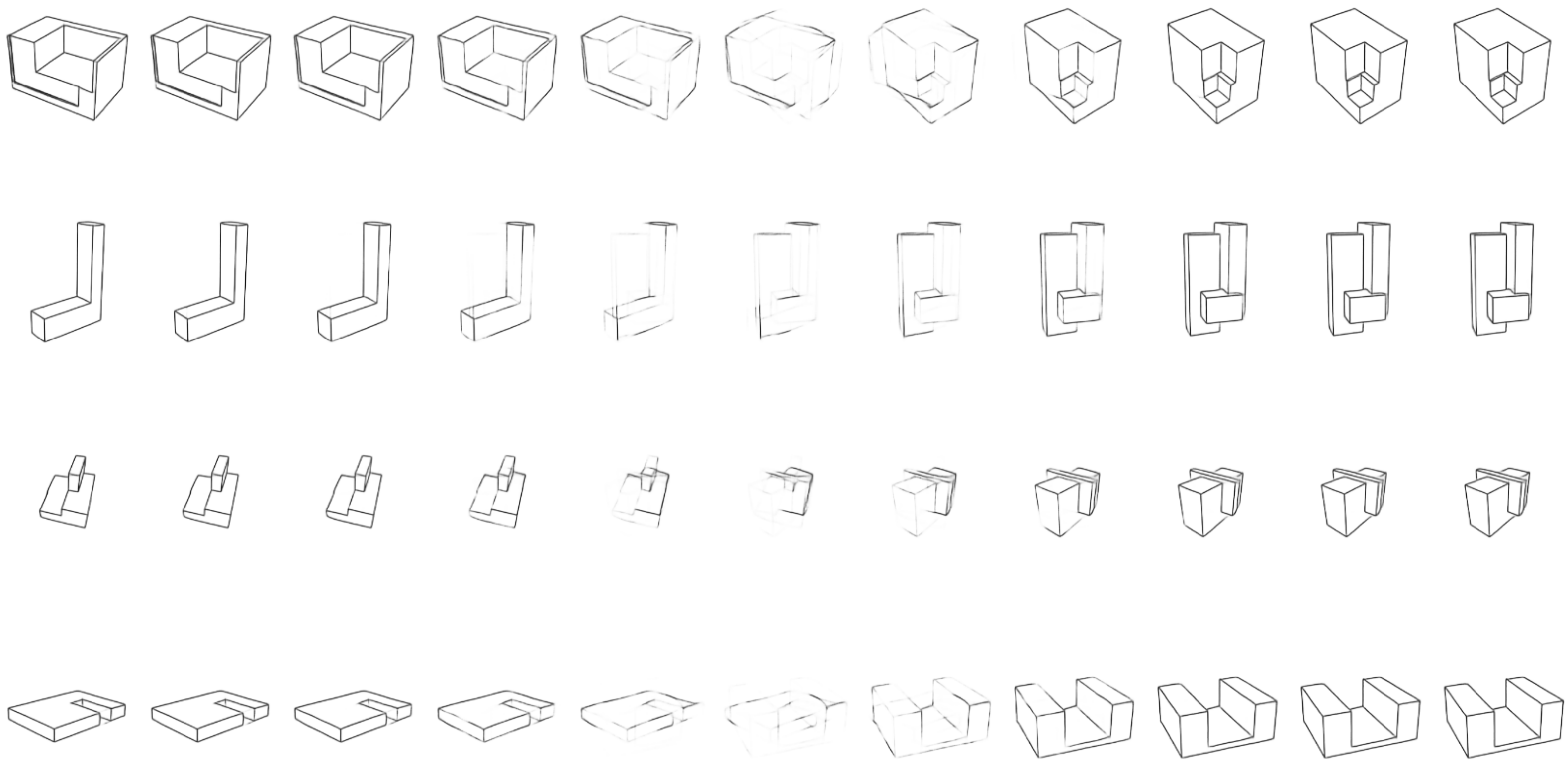


Figure 7.1 Interpolation studies on the learned latent space. The failure of the decoded interpolated vectors to “morph” between the two endpoint sketches indicates that the latent space is sparsely populated by examples, and between them lies a great deal of empty space. Two potential ways to resolve this problem are 1) lowering the dimensionality of the latent space, or 2) broadening the number and types of sketches included in the latent space.

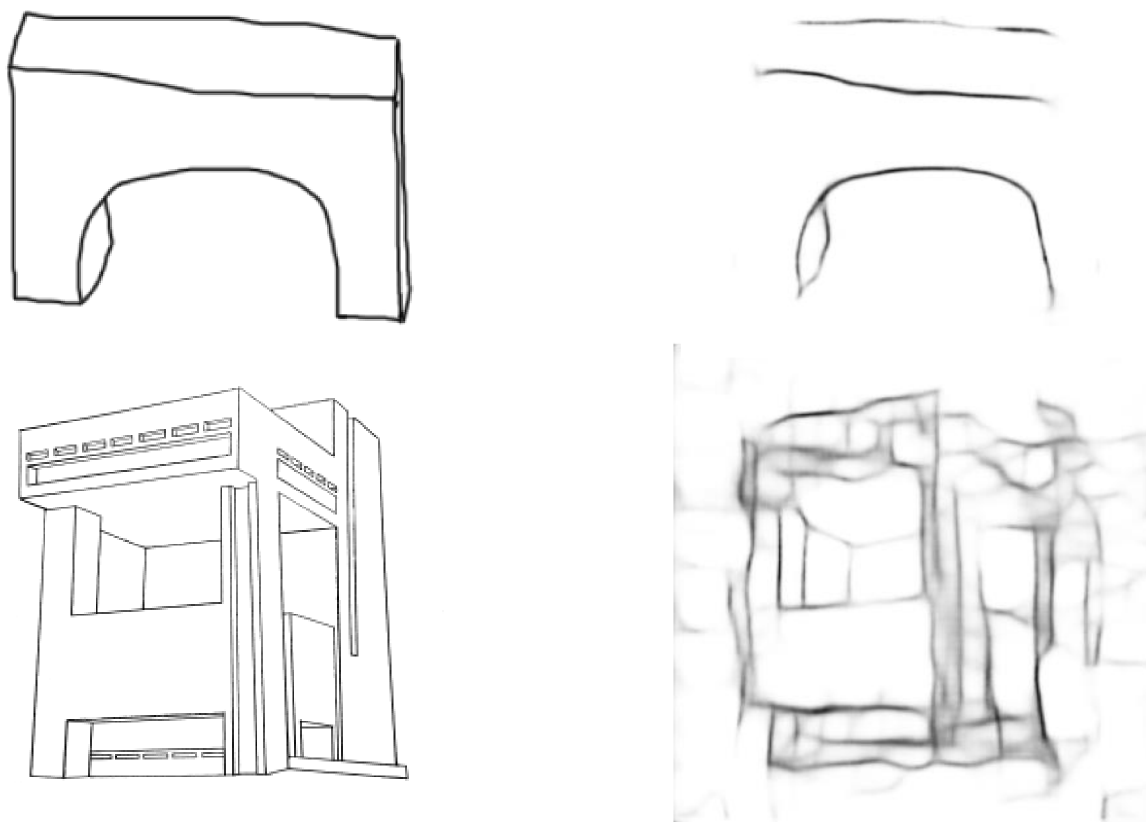


Figure 7.2 Examples of sketches with high reprojection error in the learned latent space. (Top) The generated sketch examples all centered the object in the sketch, and so the learned encodings often fail to capture details near the edges of the sketch, as can be seen by the loss of this information in the reprojection. (Bottom) The generated sketch examples all have identical line thickness, and so sketches that do not match this line thickness are heavily distorted when reprojected from the latent space.

Taking into consideration the various problems identified above, I'd like to propose what I feel would be a practical latent space for this problem. As stated above, the latent space would need to have encodings for 3D objects clustered based on their 2D projection from a specified viewpoint. To achieve this, the latent space could be trained with a 3D object-based autoencoder, but with a loss function that penalizes encodings based on both encoder-decoder reprojection error and *the latent encoding's distance in the latent space from other encodings with similar or identical 2D projections*. This would directly incorporate the goal of minimizing the reconstruction error into the latent space training as was done in [Park19] while also enforcing the desired clustering behavior. Finally, a separate encoder would be trained to generate the latent space distributions of sketch inputs.

With a sketch-encoder that generates a distribution of possible objects rather than a single object, a user could, if desired, repeatedly sample the distribution of their input sketch until finding an acceptable output, or the user could use the mean of the distribution for consistency. Furthermore, the network could be adjusted to be optionally recurrent, similar to the network presented in [Choy16]. Additional sketches from different viewpoints would create a narrower latent space distribution, until eventually the distribution was narrow enough that it described a single object. Figure 7.3 and Figure 7.4 show a visual representation of sketch distributions for the proposed latent space for both the single sketch and multiple sketch cases.

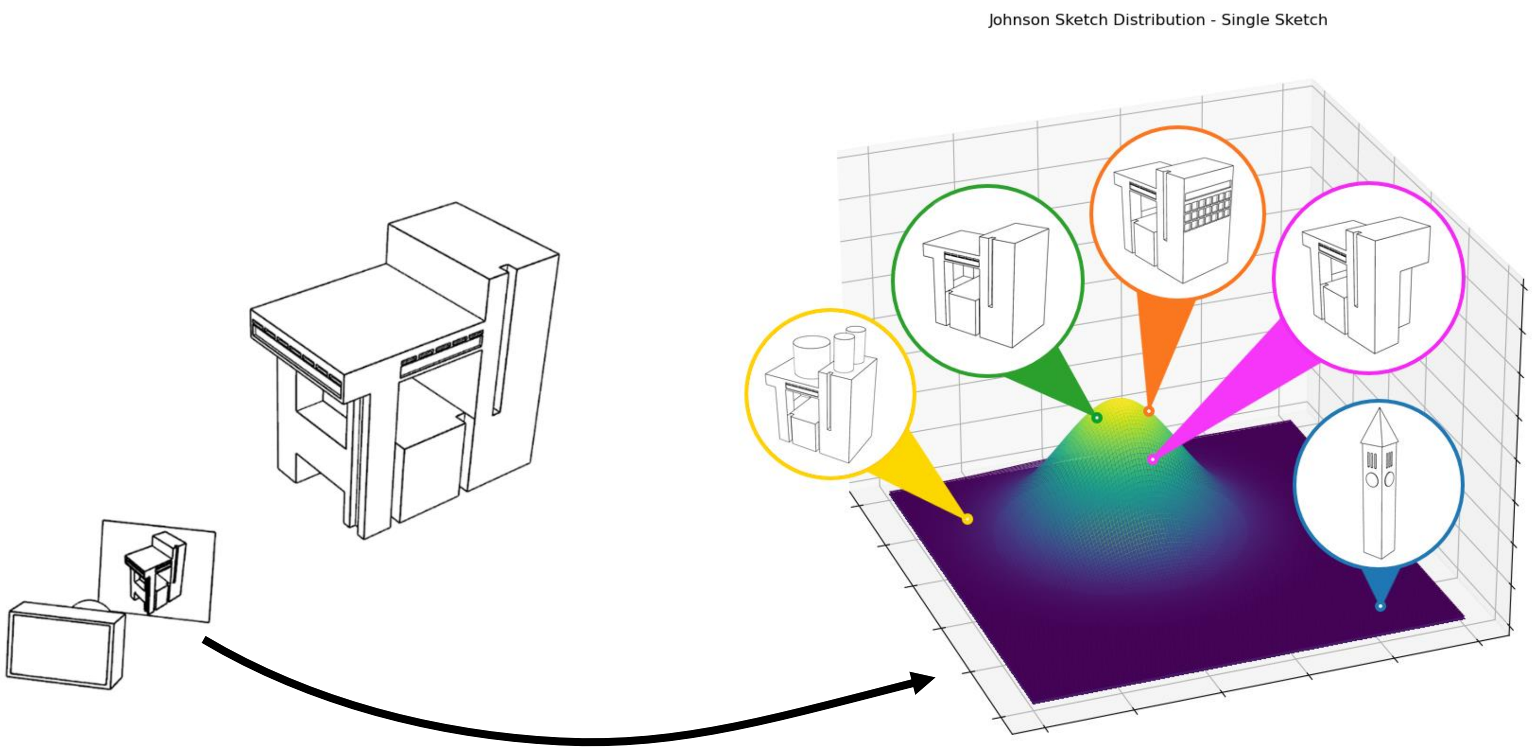


Figure 7.3 A distribution in the proposed view-dependent latent space for a single sketch input. (Left) An ambiguous sketch input that could refer to one of many possible three-dimensional objects, as it does not include information about the backside of the object. (Right) The latent space distribution for the sketch that covers all these possibilities. Note that two of the five geometries do not correspond to the sketch, and because the latent space is organized by the 2D projections of the objects, they will be outside the sketch distribution for an appropriately small variance as contrasted to the model-based latent space shown in Appendix F.

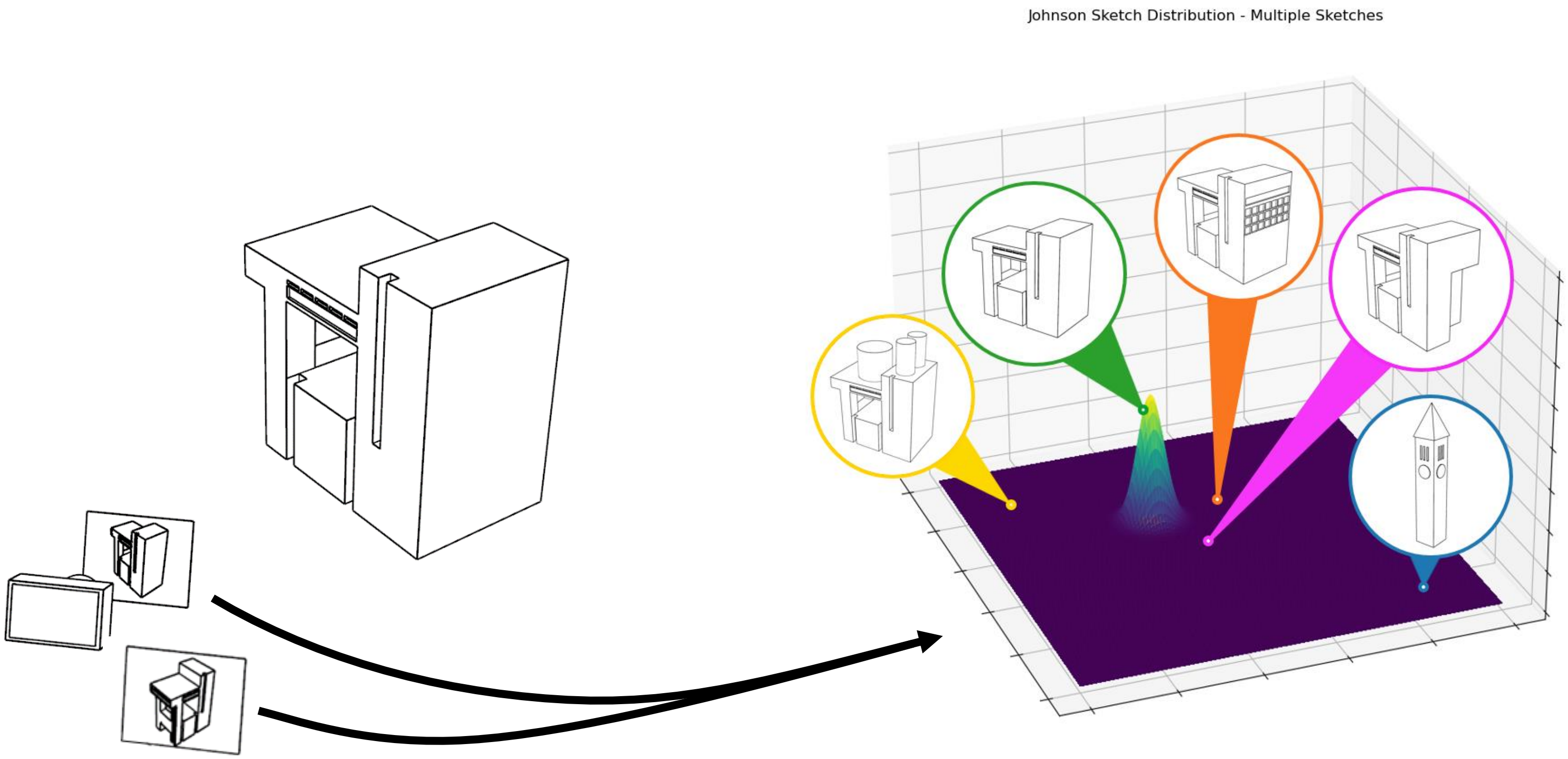


Figure 7.4 The distribution in the proposed view-dependent latent space with an additional sketch input. (Left) An additional sketch input that removes some of the ambiguity about the backside of the object. Note that some ambiguity remains as both sketches still do not include all the information about the object. (Right) The latent space distribution for the combination of the two sketches. Of the five geometries shown, only one corresponds to both sketches, and therefore it is the only object still within the narrower distribution. The other four objects are outside the distribution, including two that were in the original, broader distribution for the single sketch.

While it does solve the issues raised earlier, there are still problems with the latent space I have proposed. First, the same 3D object can appear markedly different depending on the viewpoint chosen, and so most objects would need to be stored repeatedly in the latent space, once for every possible viewpoint.[5] This is problematic because of the potential wasted space used by redundant object encodings. Moreover, the description of the loss function is high-level, and it is unclear how to develop a specific implementation for it. Even setting aside the implementation, it is possible that the reprojection error and the latent space distance components of the loss function will antagonize one another instead of guiding the learning process to a well-behaved latent space. Finally, in the case of multiple sketches, generating a narrower, refined distribution would require that all the possibilities that correspond to the second sketch form a smaller cluster in the latent space. The proposed loss function does not account for this additional requirement, and because of the vast number of possible additional viewpoints, it does not seem feasible to modify it to include a "sub-clustering" requirement.

One final possibility I would like to address is that latent spaces may not be suited for the sketch-to-model problem. My initial reasoning for incorporating a lower-dimensional sketch-based latent space was that it enabled me to closely follow the architecture of [Park19], and while I have since discovered additional benefits that result from the inclusion of a latent space encoding, it is entirely possible that sketches are ill-suited for latent representation. [Delanoy18] achieved much better success with their reconstruction approach, and it may be that a skip connection that avoids a latent representation bottleneck is crucial to a successful deep learning approach to the sketch-to-model problem.

[5] Because the number of possible viewpoints for an object is essentially infinite, it would be impossible to truly store a copy of the object for every possible viewpoint. However, because viewpoints become negligibly different as they get closer to one another, one can imagine that a large but finite number of viewpoints would be sufficient to capture all possible sketches of the object. Given that sketches are inherently inaccurate representations, this large number of viewpoints could be even further reduced if desired.

7.2 Additional Limitations

One of the main additional limitations of this thesis is that the class of models generated for training was extremely narrow: objects that can be created from three axis aligned boxes joined with Boolean operations. This limited the capabilities of both the sketch-based VAE as well as the SDF reconstruction network. Thus, a natural extension of this work is to apply it to a broader class of objects. The model generation script could vary the numbers and rotations of the boxes that are joined instead of always using three that are axis aligned, it could include non-box primitives and randomly select from the list of primitives, it could use non CSG modeling, etc. Assuming the network is capable of generalizing to a broader class of objects, this change would significantly improve the utility of the network.

Another limitation of this work is that the final output of the process is a signed distance field (or a rendered image of the surface represented by the field) and not an explicit model representation. Fortunately, a signed distance field can be converted to a mesh with a variety of different algorithms, the simplest being the Marching Cubes algorithm (see [Lorensen87] for more details). If this work were to be implemented as a software tool rather than a proof of concept, it would need to include a meshing step.

As discussed in Chapter 5, the quality of the SDF reconstruction network's results needs significant improvement, and one possible way to achieve this improvement is to drastically increase the amount of training data. I was already beginning to reach the limits of my single GPU workstation, and training on the scale described in Chapter 5 would almost certainly require the use of an HPC cluster. [Park19] states that their network was trained on 8 NVIDIA GPUs for 1000 epochs, and so I would expect a continuation of this thesis work trained with a dataset on the scale of ShapeNet to need an equivalent setup.

Other aspects of the SDF reconstruction network that could be significantly improved are the architecture and the methods used for training. Unlike the latent space encoder, the architecture of the SDF network was not varied. While vast changes to the network architecture would most likely have yielded worse results than the baseline, testing minor variations to the baseline network and optimizing the structure as was done for the latent encoder would almost certainly have improved the network's performance. Moreover, it is possible that the network was being presented too difficult a challenge at once, and that performance would have been better if it were first trained on simpler tasks (for instance, reconstructions of sketches of a single box) and then trained on the full problem. This is called *curriculum learning*, and it could easily be incorporated into future work. Finally, the hyperparameters of the network were tuned manually, and so it is highly likely that the optimal hyperparameters were not found. In future work, a more thorough tuning procedure like grid search would be beneficial to the performance of the network.

Chapter 8

Conclusion

The task of interpreting three-dimensional structure from two-dimensional sketches is an open problem, and one that is made much more challenging because of the nuances of human perception. Despite the best efforts of a body of research dedicated to algorithmic-based sketch interpretation, these algorithmic approaches were unable to address all of the assumptions inherent to the later stages of the human visual system. Thus, modern research in the area has shifted to data-driven, machine learning approaches. Some of the most promising recent research on the related problem of interpreting three-dimensional structure from digital photographs relies on implicit object representations to produce smooth, high-quality results. Here we applied these same techniques to the sketch-to-model problem.

The process for generating a three-dimensional structure from a two-dimensional sketch was as follows. First, the sketch was converted to a lower-dimensional latent space representation. Second, the latent space representation was fed as an input to a deep-learning neural network that constructed a signed distance field representation of the object. Finally, this signed distance field was repeatedly queried to render the surface of the object.

The latent space representation was learned with a convolutional variational autoencoder, and several variations of the network architecture, latent dimensions, and hyperparameters were tested to identify the best-performing combination. A rough estimate of the number of latent dimensions needed for the problem was determined with Principal Component Analysis, and the appropriate dimensionality was confirmed with measurements of the reprojection error. The sketch-encoding network was trained with augmented examples, and even though the augmentations were limited to horizontal flips and small translations, the network performance showed significant improvement both quantitatively and visually.

The signed distance field generation was learned with a network modeled after the DeepSDF architecture presented in [Park19] and was trained on millions of pairs of points and signed distances. Initially, the network was trained on a small, manually inspected dataset of 200 models, but it was found that this dataset was too small to achieve reasonable results on unseen sketches. A larger, 10,000 model dataset was generated without manual inspection, and the results were significantly improved over the smaller dataset. While many of the smaller details of the objects depicted in the sketches were lost, the rough overall form was almost always maintained, even in the test examples.

The generated objects were rendered with sphere-tracing, a relatively simple ray-marching algorithm used to identify surfaces encoded in signed distance fields. Because the generated signed distance fields had limited accuracy, additional modifications were made to the basic sphere-tracing algorithm to improve its robustness to inaccurate signed-distances and extra-domain artifacts.

Finally, with the entire sketch-to-model process outlined, we looked at where the limitations were, and how they could be addressed in future work. The biggest of the limitations discussed were those of the latent space, and I proposed a high-level description for an improved sketch-model latent space for better facilitating object reconstruction. Additional possibilities for improving the process included broadening the narrow class of objects generated as training examples, adding an extra step for generating a final mesh output, increasing the size of the dataset used for training to the scale used in [Park19], using curriculum training to gradually build up the network's capabilities rather than training the network on the full problem from scratch, and improving methods for tuning the network hyperparameters. All of these factors can be readily resolved by future work on the topic.

Now that I've repeated what I had already told you, I'd like to wrap up with a discussion about the broader implications of this thesis and related work.

First, while this work had only limited success in solving the sketch-to-model problem, it is my hope that it lays the groundwork for a future full solution to the problem. By exploring the possibilities for various sketch-encoding network architectures and reporting the results of the different possibilities, I hope to inform future work seeking to develop a deep learning approach to the problem and to aid in the development of an appropriate latent space.

Second, were the sketch-to-model problem to be solved with reliable accuracy, the software would enable designers who find current computer-aided design tools tedious to focus their efforts on their designs rather than fighting with the software. It is aggravating to have a mental image of your intended design and no simple way to translate your ideas into reality, and I hope that eventually this work can be used to help alleviate those frustrations.

Finally, the ability to develop latent space encodings of various types of inputs with deep learning has remarkable potential for further research. As stated in Chapter 2, one of the major motivations for using deep-learning to solve this problem is that humans' interpretations of two-dimensional sketches are influenced by the brain, and can rely heavily on understanding the nature of the entire sketch as is shown in Figure 2.2. Humans can represent three-dimensional structure in our minds (though it may be more difficult for some compared to others), but this representation format is quite different from the formats used in computer science. Granted that this opinion is almost certainly naïve, I feel that learned latent spaces are, at the very least, a step towards the neural representations formed in the brain, as the latent vector arithmetic and interpolation shown in [Ha17] and [Wu16] are markedly similar to the abstract conceptual manipulations that come naturally for humans.

Lofty ideals for them aside, latent spaces already have strong research potential in their current form. In Chapter 3, I asked the rhetorical question 'How many dimensions are needed to represent all possible sketches of 3D objects?' As we already discussed in the chapter, the question is difficult to answer and depends on your definition of the word "sketch." But for now, let's assume that we have successfully developed a latent space that, with some limits on the sketches' complexity, can encode any sketch of a 3D object without loss of information. Moreover, let's assume that the latent space contains 1024 dimensions, as it did for this thesis work, and that any latent space encoding of sketches with fewer dimensions will necessarily be missing information. By determining the minimum number of dimensions necessary to create the lossless latent space, we have determined the number of degrees of freedom, a useful quantitative measure of complexity, of a broad, subjective, difficult-to-quantify concept. To put this into a broader statement: latent spaces, if we assume that they have been generated correctly, provide an opportunity to ask quantitative questions and get quantitative answers about concepts that otherwise may have no quantitative notions. This capability of latent spaces creates entirely new possibilities for research on topics that, due to a reliance on subjectivity, have thus far remained difficult to study quantitatively, and I am eager to see where novel research will take them.

With that said, I thank you, the reader, for reaching the end of this thesis. I hope you that enjoyed reading through this work and that I was able to provide you with valuable insight into the challenging sketch-to-model problem.

Appendix A

A Sample Machine Learning Problem

The purpose of this appendix is 1) to provide a rough introduction to machine learning so that readers unfamiliar with the field can understand the terms and concepts used throughout this thesis, and 2) to illustrate by example some of the points raised in Chapter 3. To this end, over the course of this appendix, we will develop an "apple classifier" which will learn to determine whether or not an apple is tasty.

Suppose we were building a machine learning system to classify apples as "tasty" or "not tasty." We will eschew having some complex robotic interface for measuring the characteristics of apples automatically, and instead we'll have humans tabulate information about the apples. The machine will then attempt to learn how to determine an apple's "tastiness" from the entries of this table. Suppose that in our table we include the following six categories:

1. Apple Type[6]
2. Weight (in lbs)

---

[6] We will assume that all apples are one of the following types: *Red Delicious, Granny Smith, Golden Delicious, Honeycrisp, Pink Lady,* or *Macintosh*

3. Firmness (on a scale of 1-5)
4. Number of bruises
5. Holes (yes or no)
6. Worms (yes or no)

We say that these categories define the *features* of the machine learning problem, and that the "tastiness" of an apple is its *label*. The *dimensionality* of this machine learning problem is six, as there are six features being considered. Features five and six are binary features, features one, three, and four are discrete non-binary features, and feature two is real-valued. Our classifier will need to learn to rank a given apple as "tasty" or "not tasty" based on the apple's values for each of these six features.

A dataset for this problem might look something like Tables A.1 and A.2 (though ideally, a real dataset would have more than five entries). We often refer to each element of the dataset (in this case, each apple) as an *example*. Each example has a value for every feature as well as a label for whether it is tasty. Because the examples have labels, we call this a *labeled dataset*.

	Apple Type	**Weight**	**Firmness**	**No. of Bruises**	**Holes**	**Worms**
Apple 1	Red Delicious	0.35	3	4	Yes	No
Apple 2	Granny Smith	0.27	5	0	No	No
Apple 3	Macintosh	0.25	4	1	No	No
Apple 4	Golden Delicious	0.32	2	0	No	Yes
Apple 5	Pink Lady	0.41	5	0	No	No

Table A.1 Example dataset features.

	Tasty?
Apple 1	No
Apple 2	Yes
Apple 3	Yes
Apple 4	No
Apple 5	Yes

Table A.2 Example dataset labels.

To reiterate from Chapter 3, we can naively think of dimensionality as a rough upper bound on the problem complexity, and intuitively it makes sense that adding more information should make the problem more complex. However, depending on the feature space and the specifics of the problem, this notion can be wildly inaccurate. If the features specified do not give sufficient information for the problem to be accurately learned by a model, then the bound could be much higher. If the features contain redundant information, then the bound could be much lower.

To explain this first point in more detail, let us return to our apple example from earlier. Suppose there was no objective notion of "tastiness" for an apple, and instead it strongly depended on the preferences of the person consuming the apple (for example, I personally am quite partial to Granny Smith apples, but many find them too tart for their personal tastes). One way to resolve this issue would be to expand the feature space to include information about the person consuming the apples, which would add the necessary information to determine the apple's "tastiness." Doing so would increase the feature space's dimensionality, which would mean the true upper bound for the problem complexity is higher than the dimensionality of the original feature space that does not contain information about the apple's consumer. Thus, we cannot rely on the dimensionality of a feature space as an upper bound to a machine learning problem's complexity unless we are certain that the problem is fully "learnable" with the provided feature space. It should also be noted that the accuracy to which we want to learn the problem has a large influence on the problem's complexity, and so the desired performance of the algorithm should also be considered when determining whether or not a problem is "learnable" with a particular feature space.

For the second point, we again return to our apple example to clarify the explanation. Suppose it were always the case that an apple with either holes or worms in it is "not tasty," regardless of the values of the other features. We could collapse both feature five and feature six into a single feature (e.g. Holes or Worms (yes or no)) and we wouldn't be removing any information necessary to predict the apple's tastiness. Thus, while we can use the dimensionality of the original feature space as an upper bound on the complexity, we can see that the problem is actually simpler than the original feature space would suggest. Because computation becomes more difficult with an increasing number of features, it is desirable to reduce the feature space to as few dimensions as possible.

There are many ways to build a classifier for this problem, and for the sake of brevity, I won't explain them here. However, I would like to cover a few more basic concepts that are referenced throughout the thesis, so I will provide classification strategies to analyze in lieu of building a classifier from the data itself. Note that we will be using the original dataset for the discussion below, ignoring the modifications made in the previous paragraphs.

First, when training a model to learn a problem, it is often standard practice to set aside a small subset of the dataset to test whether or not the model generalizes to non-dataset examples. This is called a *training/test split*, and standard proportions for the split are 90% for training and 10% for testing. In our case, our dataset only contains 5 examples, so to ensure that we have at least one test example, we will split our dataset such that the first 4 examples are training examples and the last example is for testing only. This would give a training/test split of 80% to 20%.

Now, suppose that from our training data, we developed two potential strategies for classifying apples:

1. If the weight is below 0.30 lbs, label the apple as “tasty”, otherwise, label as “not tasty.”
2. If the apple has holes or worms, label the apple as “not tasty,” otherwise, label as “tasty.”

We can see from the table that both of these strategies classify all the training examples, Apples 1 through 4, correctly, and from just the training data, it is difficult to determine which of them is superior. However, if we apply both strategies to the test example, we see that Strategy 1 does not classify the test example correctly, whereas Strategy 2 does. Thus, we can say that Strategy 2 *generalizes* better than Strategy 1, and we therefore consider it to be the superior strategy.

Finally, it is worth noting that while a high performance on the test set is an indicator of a strategy that generalizes well, it does not *guarantee* that the model will generalize well to unseen cases, especially when the dataset is small compared to the set of all possible examples. Consider the following apple that our machine learning system might encounter in the future:

Apple X	Red Delicious	0.1	1	7	No	No

Table A.3 A hypothetical unseen apple. Most people would likely classify the apple as “not tasty” based on the features above, but we cannot necessarily expect a machine learning algorithm to do the same.

Given its small size, large number of bruises, and extremely low firmness, we would likely expect that the correct label for Apple X is “not tasty.” However, because there are no holes or worms, our current strategy would label this apple “tasty,” and some poor, overly trusting soul could be led astray by the incorrect conclusion of our “sophisticated” model with 100% test accuracy. In practice, to avoid misleading claims about the accuracy of the model, the dataset is often split into three categories: training, validation, and testing. Each potential strategy is trained on the training set, the best strategy is selected based on performance on the validation set, and the reported accuracy of the model is based on the performance of that strategy on the test set.

This “apple classifier” problem is obviously contrived, but the ideas presented are critical to understanding real machine learning problems, including the one discussed in this thesis. For instance, in the reconstruction section of this thesis, instead of apples, our examples are points in 3D space, and instead of “tastiness,” our labels are the signed distances of the points to the surface of the object. The models are separated into a 90% training, 10% testing split to measure the neural network’s ability to generalize. The ideas are more complicated, but fundamentally, the problem is still a supervised machine learning problem, just like our apple classifier.

For more information on machine learning, there’s a plethora of articles, posts, and videos online explaining the basics, and I highly recommend taking time to educate yourself if this subject is interesting to you. Machine learning is spreading into almost every field imaginable, and even a rough understanding of the basics will vastly improve your ability to keep up with the new innovations that it will undoubtedly bring to these fields.

Appendix B

Principal Component Analysis

As the name suggests, PCA identifies the “principal components” of a dataset, which are orthogonal linear combinations of the features that maximize the variance of the dataset’s projection onto them. While that last sentence may have been easy to follow for those with mathematical or machine learning backgrounds, for those without those backgrounds it merits further explanation. Suppose we have a dataset containing points like those shown in Figure B.1.

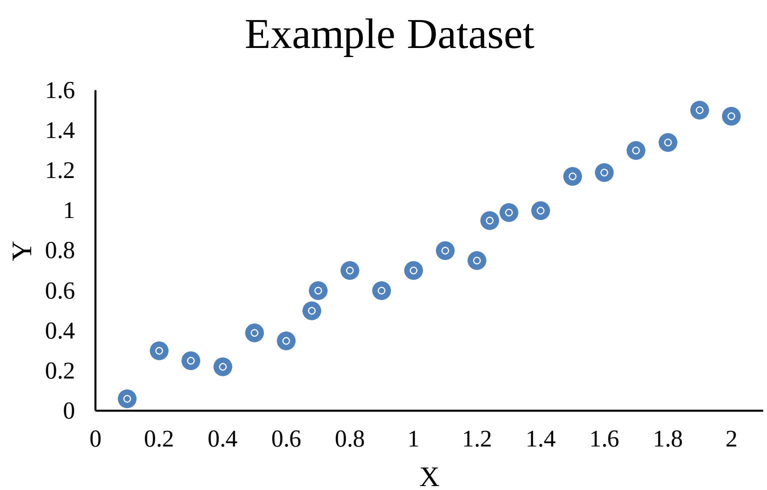


Figure B.1 An example 2D dataset.

As stated above, PCA identifies the linear combinations of the features that maximize the projected data's variance. Here we have two features, X and Y, and a linear combination of those features is just a line. Thus, we want to find the line that maximizes the variance of the projected data, or the difference between the projected data points and their arithmetic mean. An illustration of an application of PCA to the dataset is shown in Figure B.2.

The gray dashed line shows the first principal component of the dataset. One might be tempted to think of this as a "line of best fit," but there is a subtle and important difference between the two. A line of best fit attempts to minimize the residuals of the points, which is the distance in *only the y-direction* between the line and the original points. The first principal component, however, is found by maximizing the variance of the projected points (shown as black X marks), a different optimization criterion which here amounts to minimizing the *orthogonal distance* between the line and the original points.

Figure B.2 also shows the second principal component as the lighter orange dashed line. Because our dataset is already represented by two-dimensions, if we project it onto a space defined by the first two principal components, we won't lose any information about the data. However, we also won't reduce the dimensionality of the representation.

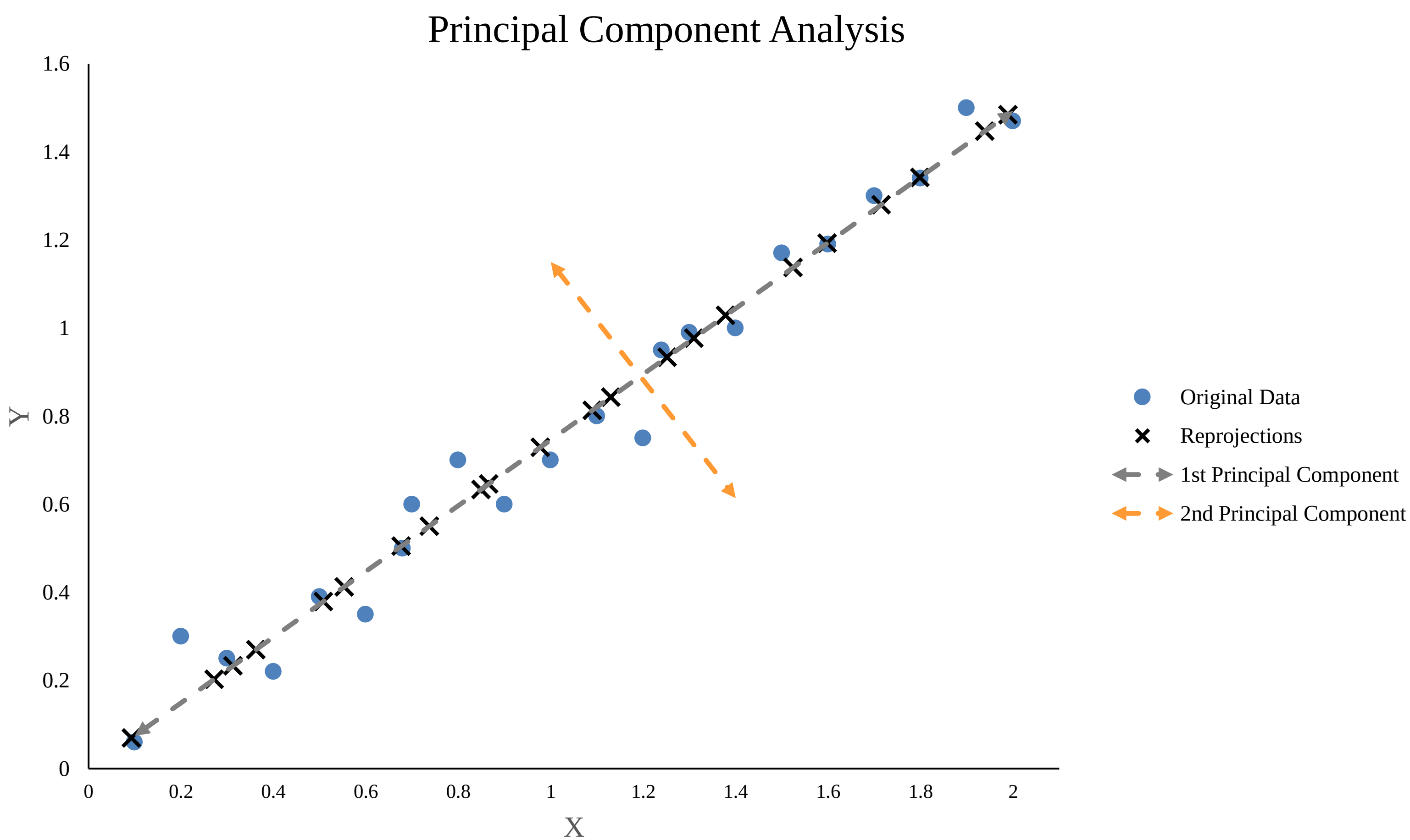


Figure B.2 An illustration of PCA applied to the example dataset. The first and second principal components are shown as grey and orange dashed lines respectively, and the black X marks represent the projection of the dataset onto the first principal component.

Even with this explanation, it is not obvious how PCA can be applied to images to create the "Eigensketches" shown in Chapter 3, but the process was relatively simple. The 256x256 image was flattened to create a 65,536-dimensional array, and each of the dimensions of the array was treated as an "axis" for PCA. The principal components are then found with a higher-dimensional analogue to the two-dimensional process shown in Figure B.2.

One of the issues with principal component analysis not addressed in Chapter 3 is that it restricts the components to linear combinations of the original features. Thus, if a particular dataset has a nonlinear lower-dimensional representation, the standard linear PCA will not find it. For example, consider the dataset shown in Figure B.3. This dataset lies on the ellipse with its major axis along the x-axis and the minor axis along the y-axis. The first principal component found with linear PCA leads to a great deal of lost information, as can be seen by looking at the difference between the original data and the projected data.

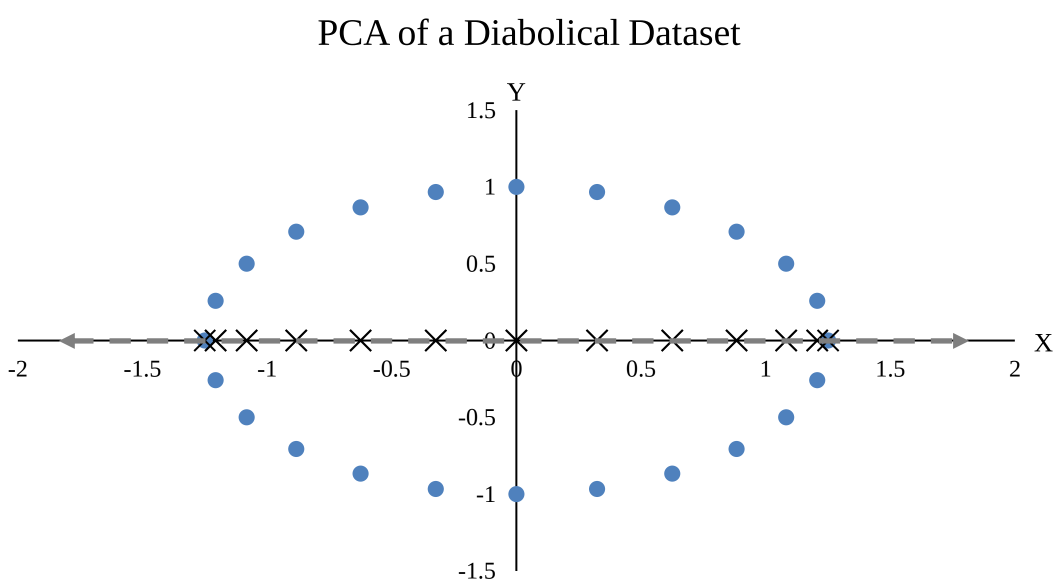


Figure B.3 A dataset which lies on an ellipse, a non-linear one-dimensional curve. The gray dashed line lying along the x-axis is the first principal component of the dataset using linear PCA, and black X marks are the projections of the dataset onto this principal component.

However, it *is* possible to reduce the dataset to one dimension, an angular parameter *t*. We can write the parametric equations for an ellipse as the following:

$$x = a\,cos(t),\;\; y = b\,sin(t)$$

In our case above, a = 1.25 and b = 1. We can solve for *t* for each dataset point, and use the values of t as our one-dimensional representation. We can then use the equations above to re-project our representation back into the original coordinate space with no reprojection error. Thus, the dataset can be losslessly reduced to one dimension using a transformation that cannot be found with linear PCA.

One way to find nonlinear principal components with PCA is to transform the dataset such that the desired principal components are linear in the transformed space. For instance, in our example above, we could define the transformation:

$$\phi(x,y) = cos^{-1}\left(\frac{x}{a}\right) \;\; if\ y\ \geq 0$$
$$\phi(x,y) = 360 - cos^{-1}\left(\frac{x}{a}\right) \;\; if\ y\ < 0$$

Note that I have chosen to represent the angular parameter *t* in degrees, and so the inverse cosine terms should be evaluated with degrees, not radians.

Our dataset would then be in a new, one-dimensional space, and performing PCA on the transformed data would give a principal component that corresponds to the ellipse on which the original data lies.[7]

In practice, this type of approach is implemented as *kernelized PCA*, which is slightly different from the approach described above in that the transformed dataset examples are never explicitly computed. An in-depth derivation of kernelized learning

[7] It is largely pointless to run PCA in a one-dimensional space, as the first principal component covers the entire one-dimensional space. However, PCA is useful when the transformed data exists in a space with two or more dimensions.

and how it applies to PCA is beyond the scope of this thesis, but it should be noted that this allows for fast computation of PCA for high-dimensional transformations.

One commonly used kernel, the Radial Basis Function kernel, represents a transformation into an infinite-dimensional feature space, and it can theoretically represent any desired behavior. However, my attempts at implementing it into the sketch analysis were less than successful. Figure B.4 shows an example of a sketch and the reprojection after using kernelized PCA with the RBF kernel.

Figure B.4 An example sketch from the PCA dataset and its reprojection from the principal components found with the radial basis kernel. All sketch reproductions from the RBF PCA sketch space looked nearly identical to the one shown above, regardless of the number of principal components used to define the space.

The right-hand image from the figure above was generated with the exact same code as the examples generated with linear PCA shown in Chapter 3; the only parameter that was changed between these two runs was the type of kernel used. Given the spectacular failure of this approach and the clear potential of learning a low-dimensional representation with a variational autoencoder, I decided not to pursue PCA-based approaches any further.

Appendix C

Variational Autoencoders

Autoencoders are neural networks that have been designed to compress the information needed to represent a set of possible inputs. Figure C.1 shows a representative structure of an autoencoder.

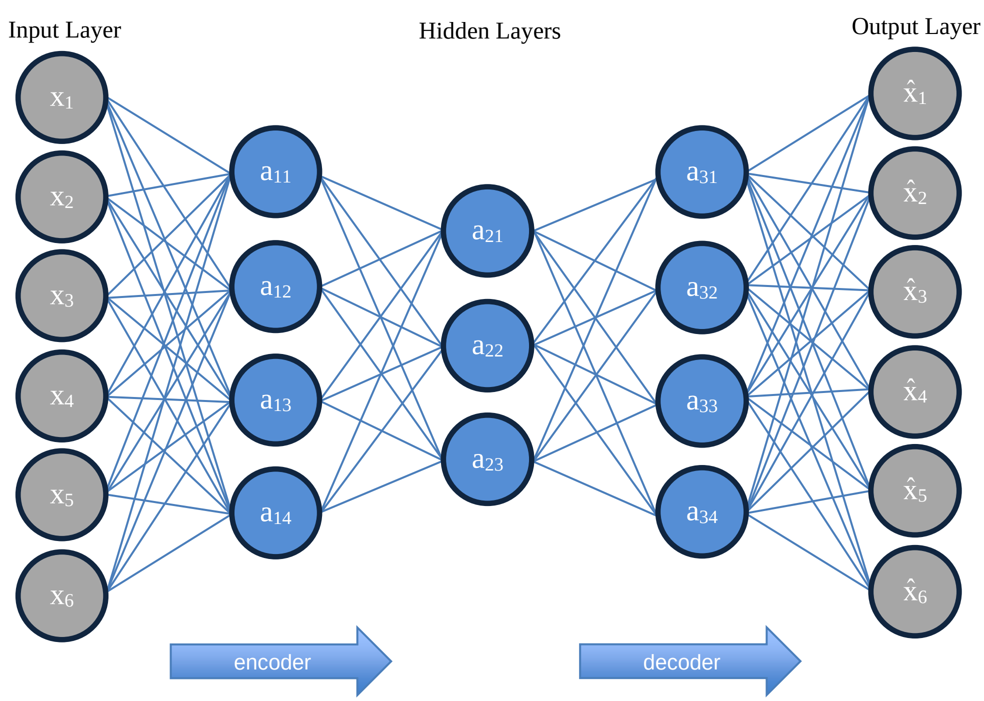


Figure C.1 A representative diagram of an autoencoder (recreated from [Jordan18a]).

We can think of this network as two separate parts on either side of a low-dimensional "bottleneck layer," shown above as the three-neuron hidden layer in the middle of the diagram. The section of the network preceding and including this layer is called the *encoder*, and it can be thought of as "encoding" the original input into a lower-dimensional representation called the *latent representation*. The section of the network following this layer is called the *decoder*, and it can be thought of as "decoding" the latent representation back into the format of the original input. We can train the network by adjusting the network's parameters to minimize the difference between an input and the network's output for that input.

The basic form of an autoencoder is prone to overfitting, and thus, many different modifications to the original framework have been developed to avoid overfitting. [Jordan18a] provides an excellent description of quite a few of these and of the overall topic of autoencoders. The adjustment I chose was to use a variational autoencoder, which is described very well in both [Rocca19] and [Jordan18b]. To summarize briefly, while a standard autoencoder converts a given input to a vector with one dimension for each neuron of the bottleneck layer, a variational autoencoder converts an input to a multi-dimensional probability distribution where each neuron in the bottleneck layer represents either a mean value or a variance in a particular dimension. Thus, for an n-dimensional latent space, the variational autoencoder's bottleneck layer must have $2n$ neurons. During training, a given example is never transformed directly to a latent vector; instead, it is transformed into a probability distribution, and a random latent vector is drawn from the distribution. This modification helps ensure that the latent space is smooth and reduces the likelihood of the model overfitting.

Because the input to the sketch-to-model process is a rasterized image, a convolutional neural network seemed the appropriate choice. Thus, the network I chose for training is a Convolutional Variational Autoencoder (sometimes abbreviated as CVAE, though this acronym can also be used to mean "Conditional Variational

Autoencoder”). Figure C.2 shows the reconstruction of a sample test input over the course of training.

It is clear from Figure C.2 that the network improved over time, but how did it learn this improvement? Like any other neural network, variational autoencoders learn by trying to optimize an objective function, and in our case, the objective function is called the Evidence Lower Bound (ELBO). The mathematical foundations and derivation of the ELBO function are outside the scope of this thesis work, and for the purposes of understanding this thesis, it is sufficient to know that a variational autoencoder’s objective is to bring the ELBO as close to zero as it can. Because the ELBO is always negative (or zero), this is equivalent to maximizing the ELBO. Thus, the networks selected as the highest performing were those with the *least negative* ELBO values. Plots of the VAE networks’ ELBO for a test set input are shown in Figure 4.1, Figure 4.4, Figure 4.6, and Figure 4.8, and the increases in ELBO of the curves shown in these plots roughly correspond to improvements in the visual quality of the reprojections.

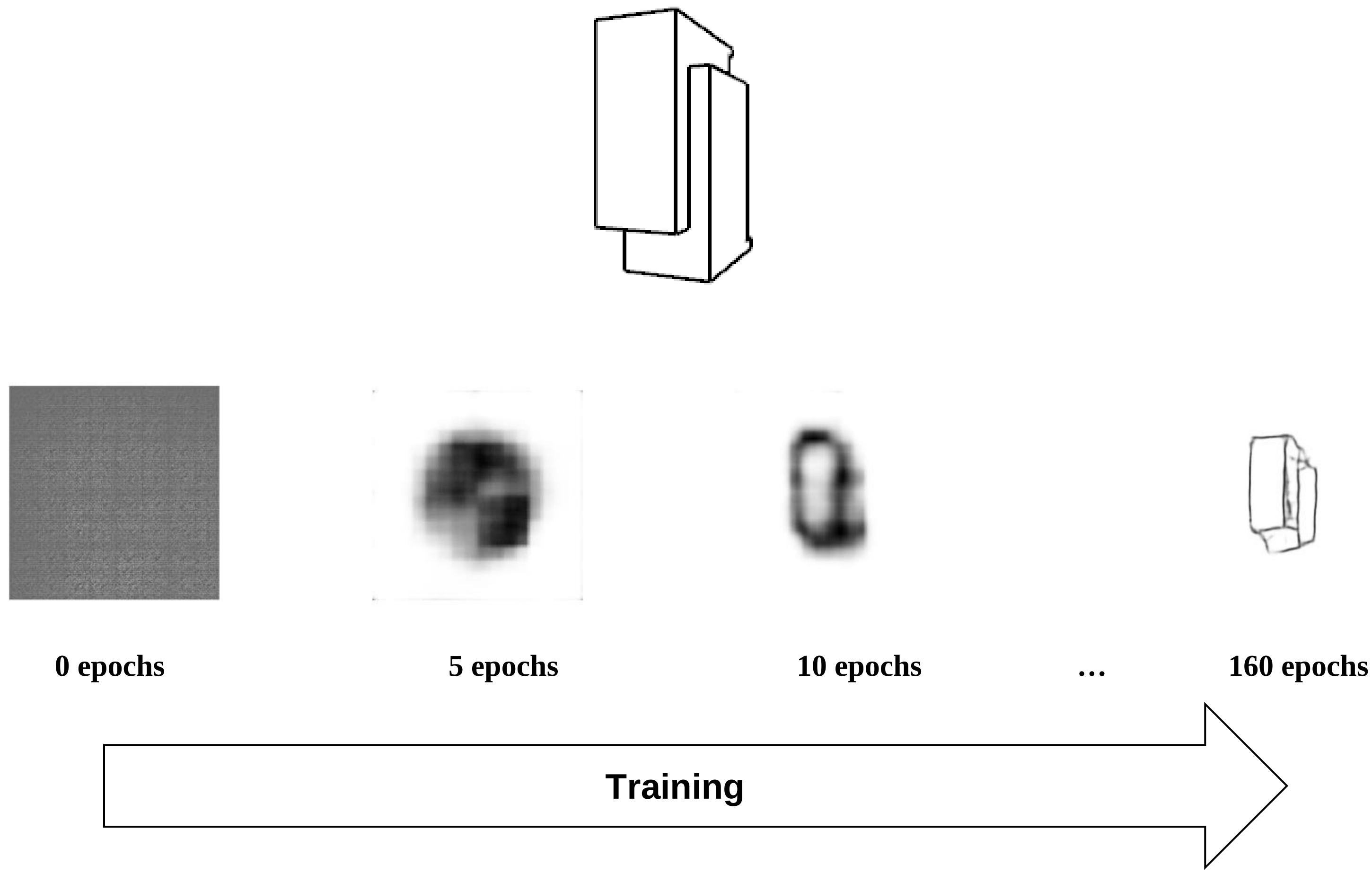


Figure C.2 An overview of autoencoder training. (Top) A sample test input to an autoencoder. (Bottom) A timeline of the autoencoder's reprojections of the sample input.

Appendix D

Limitations of Data Augmentation

In Chapter 4, after the discussion of the various architectures that were tested, I define *data augmentation* and describe two potential shortcomings it has. The first is that simple augmentations of existing examples may leave out important information about the set of possible inputs to the machine learning problem, and the second is that the augmentations can be harmful to learning if they generate examples that are outside the set of possible inputs.

To illustrate the first point, consider the case where our machine learning input is the set of all possible images of rectangles, and we have one training example as shown in Figure D.1. We might consider introducing translations and rotations of the original image as augmentations to the original dataset. Using just these two augmentations, we could easily construct thousands of training examples, a few of which are shown in Figure D.2. However, because our only training example is an image of a square, these two augmentations cannot generate examples of rectangles with uneven sides, and so they fail to include crucial information about the set of all possible inputs. This principle is also the most reasonable explanation for the distorted reprojections shown in Figure 7.2.

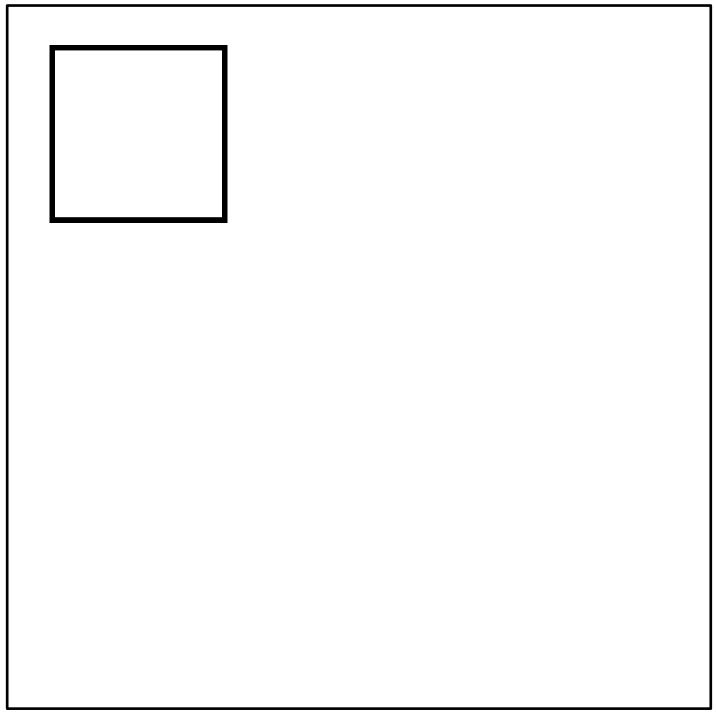


Figure D.1 A single image of a square, which we will use to illustrate the potential dangers of data augmentation.

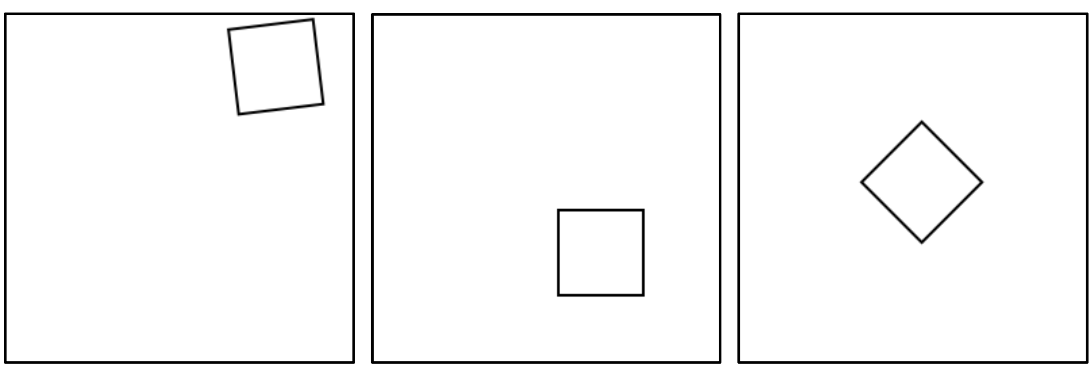


Figure D.2 Augmented examples of the single training image using translational and rotational augmentations. Note that while we can move the square to all locations in the image and specify any rotation, we cannot create images of rectangles with just these augmentations.

To illustrate the second point, suppose we were to include a non-uniform scaling augmentation to try to create examples of rectangles, as shown in Figure D.3. We can see that the borders of the rectangles have also been scaled, resulting in uneven border thickness. This is not the way we would typically draw rectangles, and thus, it can be argued that these examples are outside of the set of possible inputs we want to learn. One can imagine that in more drastic cases, these kinds of examples could harm the learning algorithm's performance rather than helping it.

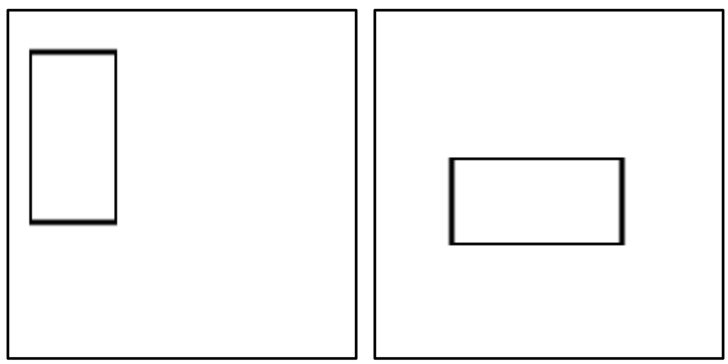


Figure D.3 Scaling-augmented examples of the single training image. Because of the distortion to the line width, these augmentations generate examples outside what we may consider "acceptable" inputs.

Appendix E

Sphere Tracing Example

As promised in Chapter 6, I'd like to walk through an example demonstrating the intuition behind sphere tracing. Suppose that we have two rays as shown in Figure E.1.

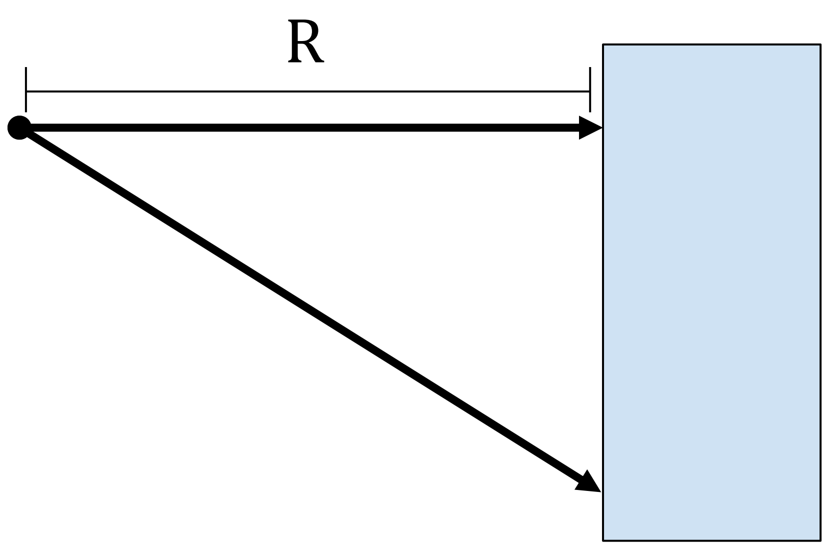


Figure E.1 Example rays intersecting the surface of an object. Both rays have the same starting point that is a distance *R* from the surface of the object. Note that the object's surface must be closed to have a proper notion of "signed distance."

While it can be formally proven that the algorithm is guaranteed to converge without overshooting for a ray pointed at a surface, here I simply want to provide an intuitive basis for reaching that conclusion without the rigors of a proof. There are two possibilities for a ray if it intersects a surface: 1) the ray points directly at the closest surface, or 2) the ray does not point directly at the closest surface. The two example rays in Figure E.1 represent these two cases.

Now suppose that the point at the far left of both rays is the starting point for sphere tracing and that R is the distance from that starting point to the surface. Because the two rays share the same starting point, the distance we move along both rays is R. We can easily visualize this by creating a sphere with radius R around the starting point and stepping to the point where each ray intersects the sphere as shown in Figure E.2. The top ray is parallel to the object surface, and if we take one step of distance R along the ray, we reach the surface. This is the quickest possible convergence for a ray. The lower ray is not parallel to the object surface, and if we take one step of distance R along the ray, we move to the point indicated in green in the figure. From here, a new sphere is created based on the new signed distance, and the process repeats until the position along the ray is within a small threshold distance of the surface, at which point it is considered to have “reached” the surface.

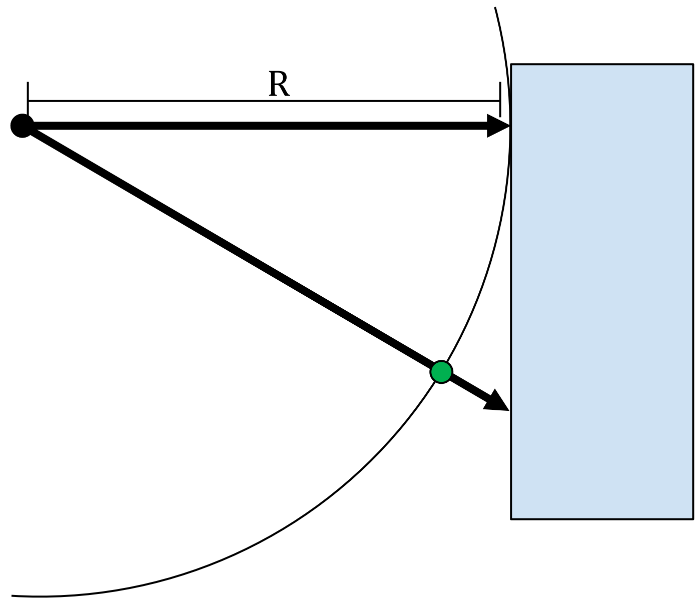


Figure E.2 The first iteration of the sphere tracing algorithm. The position along the lower ray steps out to the intersection of the ray and the sphere, shown in green.

This process is simple and guaranteed to eventually converge for rays that intersect a surface, but it does unfortunately lead to cases where certain rays require large numbers of iterations to reach the surface. Examples of problematic behavior are shown in Figure E.3.

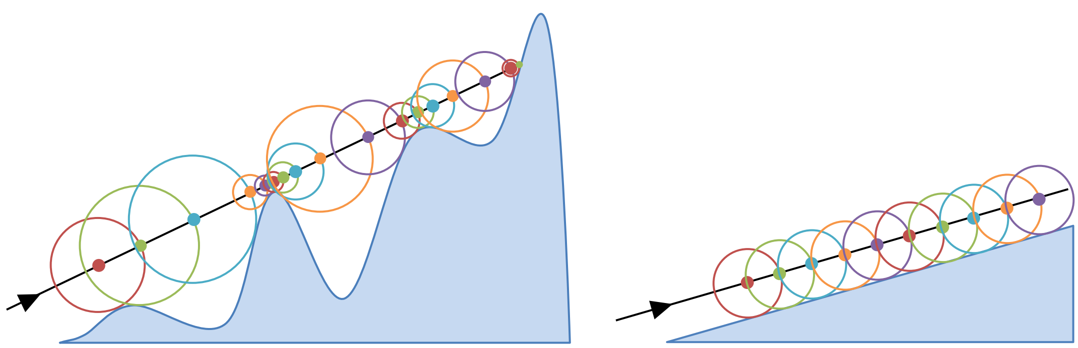


Figure E.3 Problematic edge cases for the sphere tracing method, recreated from [Takikawa22]. (a) The ray experiences multiple near-misses, which result in many extra iterations before the ray approaches the surface. (b) The ray is near the surface and parallel to it, which leads to a large number of iterations with small step sizes.

Appendix F

Model-Based Latent Space Limitations

In Chapter 3 and Chapter 7, I promised to explain my hesitation towards latent spaces that are organized based on the geometry of the 3D objects being represented. Here I fulfill that promise.

Consider the sketch shown at the top of Figure F.1. The backside of the object depicted could have any number of possible geometries - two of which are shown at the bottom of Figure F.1. As stated in Chapter 4 and Appendix C, a variational autoencoder represents a given input as a *distribution* of possible latent encodings, and so it might be desirable for our latent space encoder to output a distribution of all the possible 3D objects corresponding to a given sketch. However, this is exceedingly difficult if the latent space encodings are based on the 3D geometry of the object without accounting for the specific viewpoint at which the object is viewed. One can imagine that the drastically different geometries of the two objects shown at the bottom of Figure F.1 will lead to drastically different encodings in a model-based latent space. However, because the variational autoencoder assumes the latent distributions are Gaussian, the possible objects corresponding to a particular sketch need to be clustered together. Otherwise, the distribution will not be able to include all possible objects corresponding to the sketch without also including objects that do not correspond to the sketch. This problem is illustrated visually in Figure F.2, Figure F.3, Figure F.4, and Figure F.5.

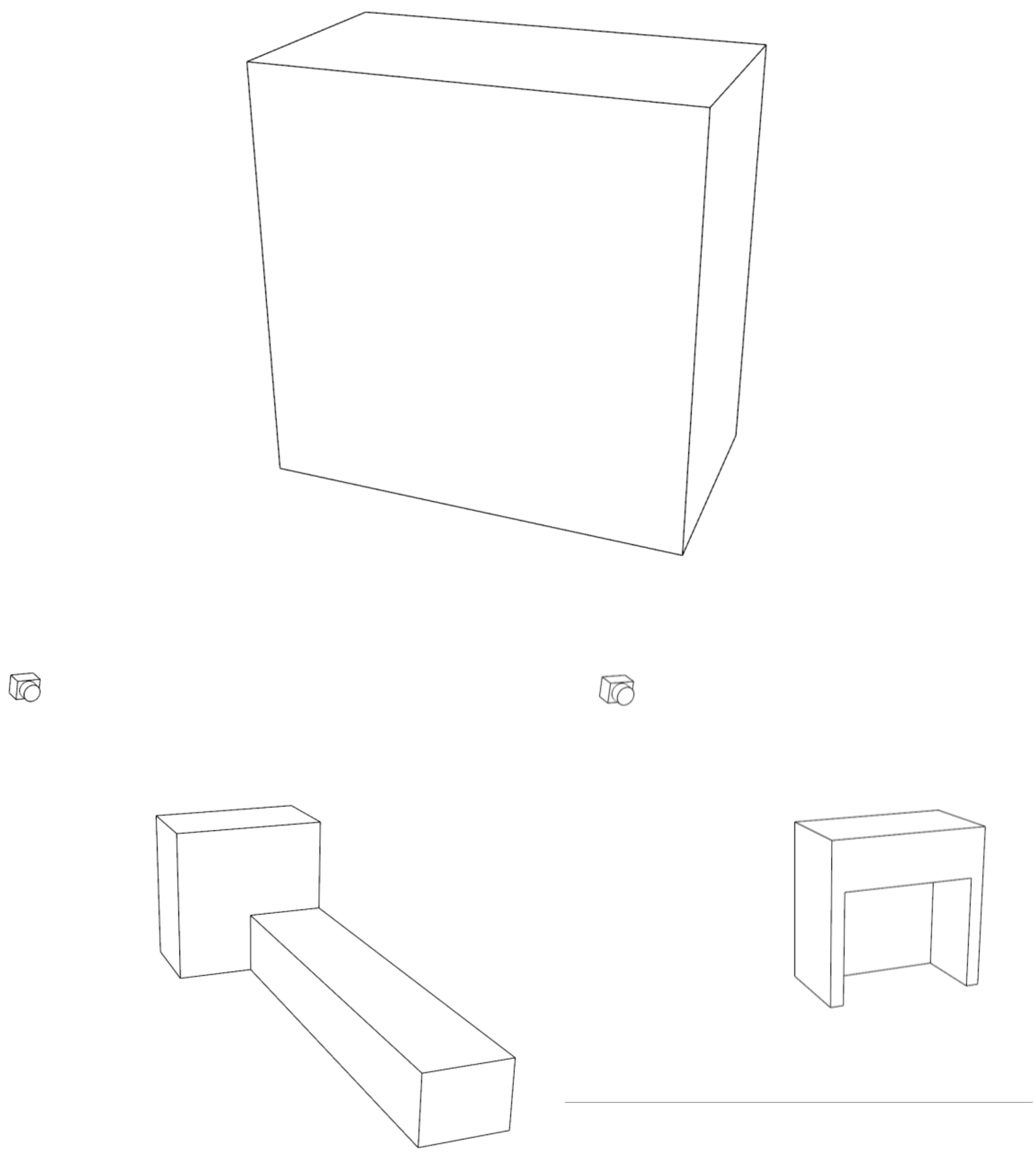


Figure F.1 An example sketch of a 3D object (Top) and two vastly different geometries (Bottom Left, Bottom Right) that correspond to the sketch. The camera line drawing indicates the rough camera position required to generate the sketch shown at the top for each of the two objects.

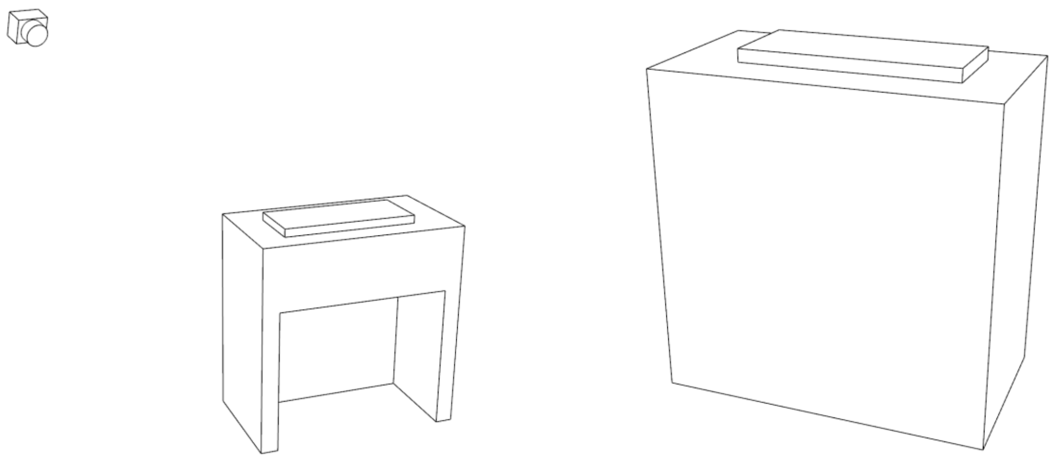


Figure F.2 (Left) An additional example object that is geometrically similar to the object shown in the bottom-right of Figure F.1 but is *not* a possible object for the sketch shown at the top of Figure F.1. (Right) A sketch of the object generated from the viewpoint of the camera shown in the left-hand side of the figure.

The object shown in Figure F.2 is geometrically similar to the object shown in the bottom-right of Figure F.1, and so we would expect both of these objects to have similar latent representations in a model-based latent space. Keeping this in mind, Figure F.3 depicts an example of a possible set of encodings in a model-based latent space for the three objects shown in Figure F.1 and Figure F.2.

Now, suppose that we wanted to use this latent space in our sketch-to-model pipeline. We would therefore need to train an encoder to convert sketches into encodings of the possible corresponding 3D objects. Assuming we still wanted to use a variational autoencoder, the encoder would need to generate a distribution that 1) includes all the possible 3D objects that correspond to the sketch and 2) does not include objects that do not correspond to the sketch. Figure F.4 and F.5 show that this is not possible with either low-variance or high-variance Gaussian distributions.[8] It is possible to use an

[8] Figure F.4 and F.5 show distributions with uniform variance across all dimensions, but the output distributions of variational autoencoders are multivariate Gaussian distributions, meaning that they can have different variances in each dimension. Thus, the encoder can 'squish' and 'stretch' the distribution in each dimension to try to exclude non-

encoder that generates distributions other than Gaussians, but unless there exists a pattern in the latent space that connects objects corresponding to a given sketch, it will be exceedingly difficult to train an encoder to generate the latent encodings of the objects corresponding to a sketch.

corresponding objects, making it more robust to the issues caused by geometry-based clustering than is demonstrated in the figures. However, because there are many more 3D objects that do or do not correspond to a given sketch than there are dimensions of the latent space, it is reasonable to conclude that this per-dimension variance flexibility alone will not resolve the issues raised in this appendix.

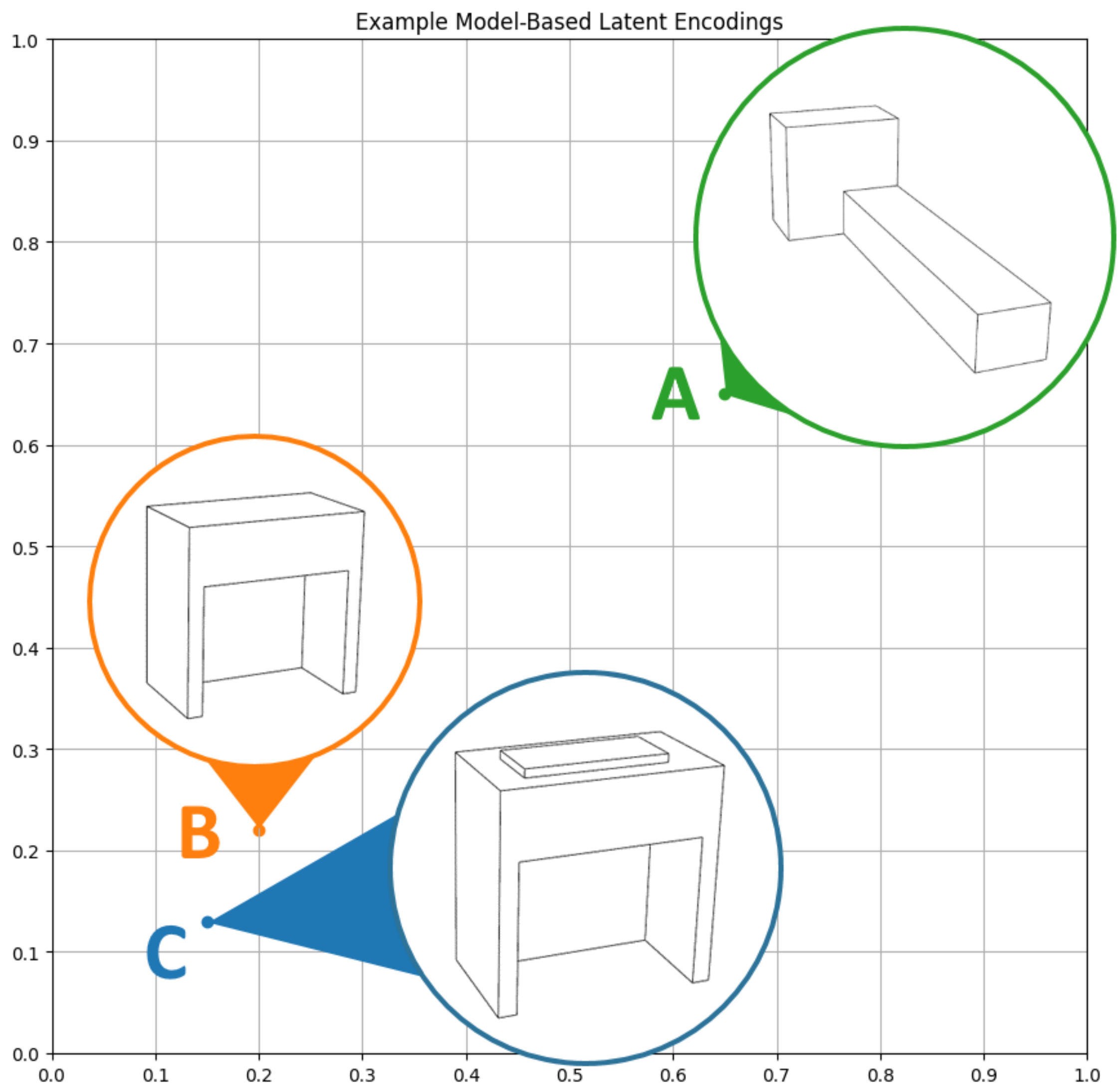


Figure F.3 A hypothetical example of encodings for the three objects in a model-based latent space. Note that because of geometric similarities, the objects labeled ‘B’ and ‘C’ are in close proximity to one another in the latent space.

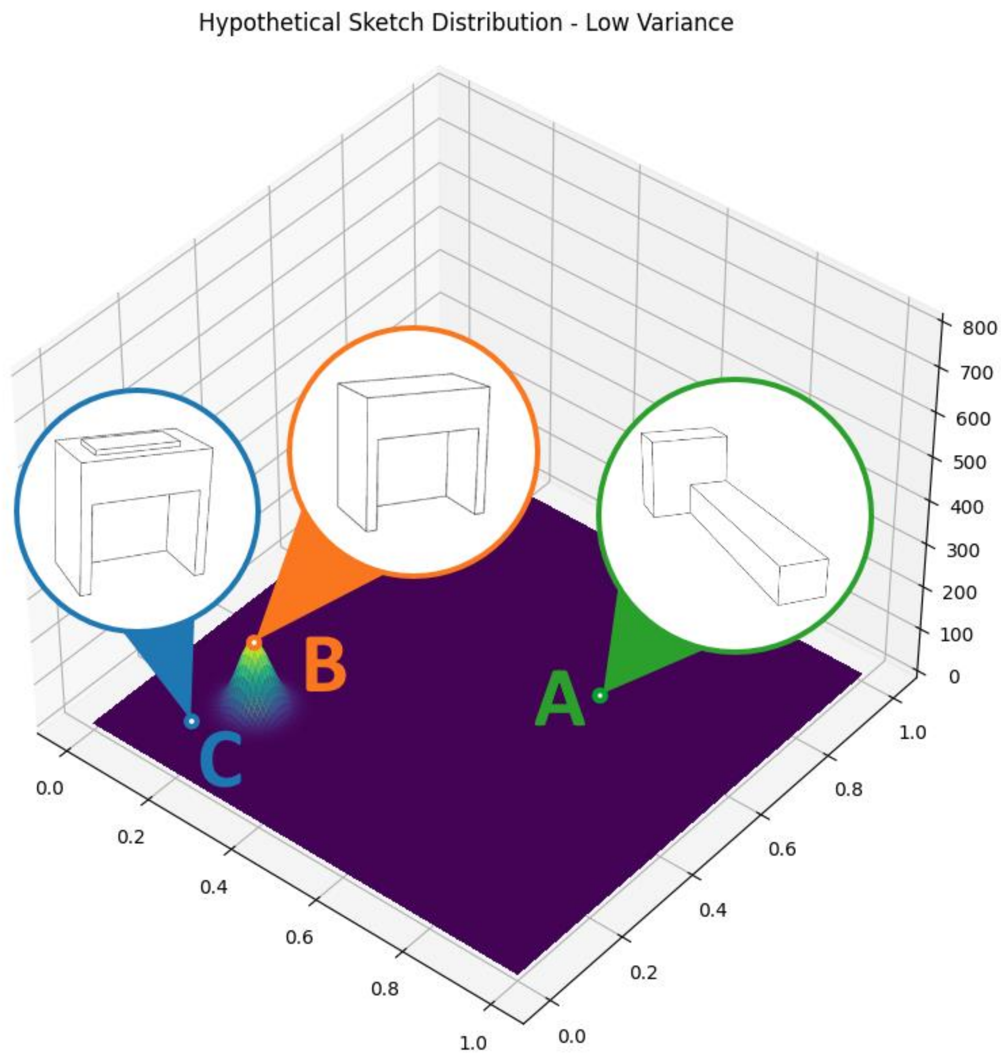


Figure F.4 An example of a low-variance distribution for encoding the sketch possibilities. Because of the distribution's small variance, it fails to include all the possible objects corresponding to the sketch (in this case, 'A' is practically impossible to draw from the distribution despite corresponding to the sketch). One can imagine that a distribution with similar variance centered on 'A' would fail to include 'B,' and that a distribution with the mean placed between these two objects would fail to include either of them.

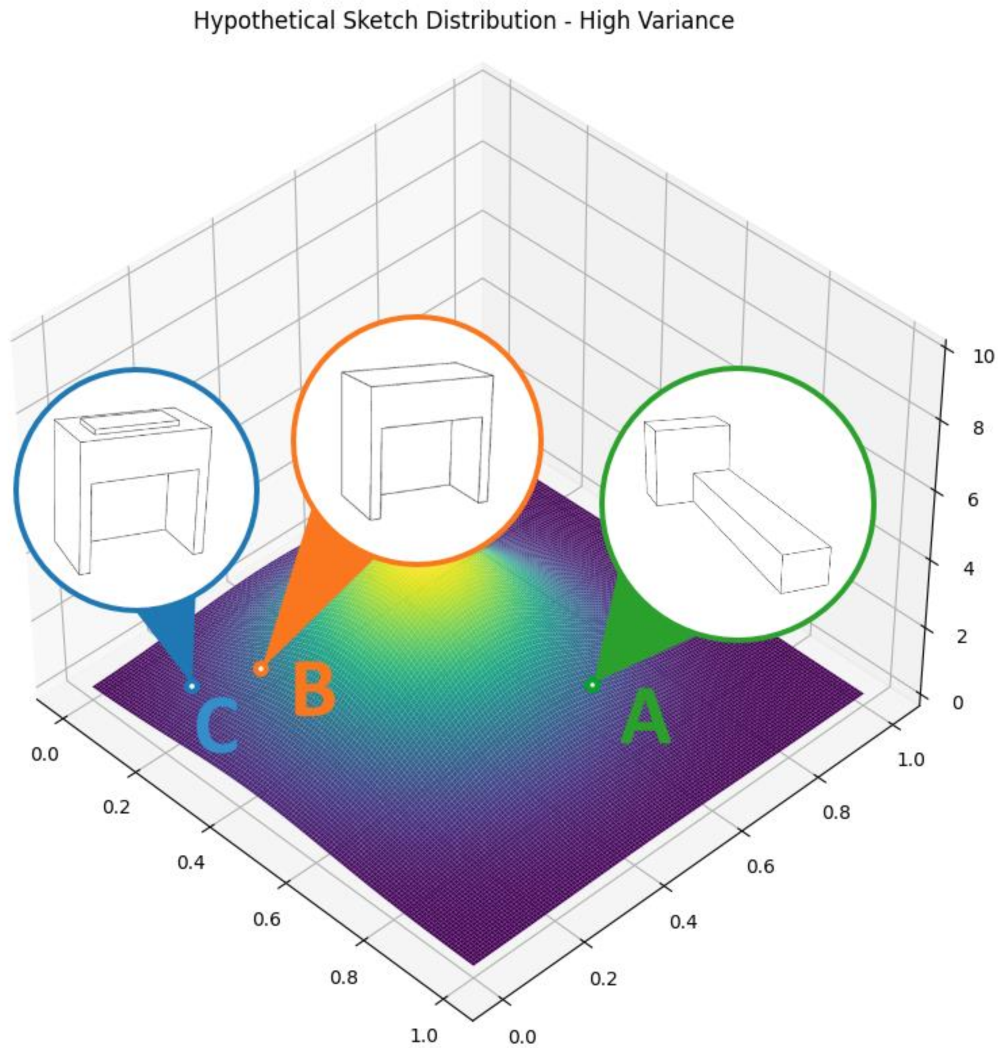


Figure F.5 An example of a high-variance distribution for encoding the sketch possibilities. Both the sketch possibilities ‘A’ and ‘B’ now have reasonable likelihoods in the distribution, but the incorrect object ‘C’ also has a reasonable likelihood. Because of the proximity of ‘B’ and ‘C’ in the latent space, any distribution with a large variance will have relatively similar likelihoods for both objects, and so it will necessarily include incorrect possibilities for the sketch.

References

[Cao05] L. Cao, J. Liu, and X. Tang, "3D object reconstruction from a single 2D line drawing without hidden lines," *Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05)*, 2005.

[Chen19] Z. Chen and H. Zhang, "Learning implicit fields for generative shape modeling," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[Choy16] C. Choy et. al., "3D-R2N2: A unified approach for single and multi-view 3D object reconstruction," *European conference on computer vision*, 2016.

[Delanoy18] J. Delanoy et. at., "3D sketching using multi-view deep volumetric prediction," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 2018.

[Fan17] H. Fan, H. Su, and L. Guibas, "A point set generation network for 3D object reconstruction from a single image," *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[Gao19] J. Gao et. al., "DeepSpline: Data-driven reconstruction of parametric curves and surfaces," *Arxiv preprint*, 2019.

[Granshaw18] S. Granshaw, "Structure from motion: origins and originality," *The Photogrammetric Record*, 2018.

[Ha17] D. Ha and D. Eck, "A neural representation of sketch drawings," *International Conference on Learning Representations*, 2018.

[Hart95] J. Hart, "Sphere Tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, 1995.

[Hill16] C. Hill, “Visualizing the Bivariate Gaussian Distribution,” *Learning Scientific Computing with Python*, 2016.

[Jordan18a] J. Jordan, “Introduction to autoencoders,” *Jeremy Jordan*, 2018.

[Jordan18b] J. Jordan, “Variational autoencoders,” *Jeremy Jordan*, 2018.

[Kang04] D. J. Kang, M. Masry, and H. Lipson, “Reconstruction of a 3D object from a main axis system,” *AAAI Fall Symposium Series: Making Pen-Based Interaction Intelligent and Natural,* 2004.

[Kingma14] D.P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *2nd International Conference on Learning Representations (ICLR 2014)*, 2014.

[Kingma15] D.P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” *3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[Li17] C. Li et. al., “BendSketch: Modeling freeform surfaces through 2D sketching,” *ACM Trans. Graph. 36, 4,* 2017.

[Lipson95] H. Lipson and M. Shpitalni, “A new interface of conceptual design based on object reconstruction from a single freehand sketch,” *Annals of the CIRP* Vol. 44/1, pp. 133-136, 1995.

[Lipson96] H. Lipson and M. Shpitalni, “Optimization-based reconstruction of a 3D object from a single freehand line drawing,” *Journal of Computer Aided Design* Vol. 28/8 pp. 651-663, 1996.

[Lipson00] H. Lipson and M. Shpitalni, “Conceptual design and analysis by sketching,” *Journal of AI in Design and Manufacturing (AIEDEM) 14,* pp. 391–401, 2000.

[Lipson02] H. Lipson and M. Shpitalni, “Correlation-based reconstruction of a 3D object from a single freehand sketch,” *AAAI Spring Symposium on Sketch Understanding*, pp. 99–104, 2002.

[Lorensen87] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” *SIGGRAPH ’87: Proceedings of the 14th annual conference on computer graphics and interactive techniques*, pp. 163–169, 1987.

[Masry05] M. Masry and H. Lipson, “A sketch-based interface for iterative design and analysis of 3D objects,” *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, 2005.

[Mescheder19] L. Mescheder et. al., "Occupancy networks: learning 3D reconstruction in function space," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* 2019.

[Park19] J. J. Park et. al., "Deepsdf: Learning continuous signed distance functions for shape representation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* 2019.

[Rocca19] J. Rocca and B. Rocca, "Understanding variational autoencoders (VAEs)," *Towards Data Science*, 2019.

[Shpitalni95] M. Shpitalni and H. Lipson, "'Identification of faces in a 2D line drawing projection of a wireframe object," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1995.

[Shpitalni97] M. Shpitalni and H. Lipson, "Classification of sketch strokes and corner detection using conic sections and adaptive clustering," *ASME Journal of Mechanical Design* Vol. 119/2, pp.131-135, 1997.

[Takikawa22] T. Takikawa, A. Glassner, and M. McGuire, "A dataset and explorer for 3D signed distance functions," *Journal of Computer Graphics Techniques,* 2022.

[Tatarchenko17] M. Tatarchenko, A. Dosovitskiy, and T. Brox, "Octree generating networks: efficient convolutional architectures for high-resolution 3D outputs," *2017 IEEE International Conference on Computer Vision (ICCV),* 2017.

[Turk91] M. Turk and A. Pentland, "Face Recognition Using Eigenfaces," *Proceedings CVPR*, 1991.

[Villaneuva09] J.C. Villaneuva, "How many atoms are there in the universe?" *Universe Today,* 2009.

[Wang18] N. Wang et. al., "Pixel2Mesh: Generating 3D mesh models from single RGB images," *Computer Vision – ECCV 2018,* 2018.

[Wu16] J. Wu et. al., "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," *29th Conference on Neural Information Processing Systems (NIPS 2016),* 2016.

[Yan17] X. Yan et. al., "Perspective transformer nets: learning single-view 3D object reconstruction without 3D supervision," *Advances in Neural Information Processing Systems*, 2016.

[Zhang21] S.H. Yang, Y.C. Guo, Q.W. Gu, "Sketch2Model: View-aware 3D modeling from single free-hand sketches," *Computer Vision and Pattern Recognition*, 2021.

[Zheng17] M. Zheng and M. Miller, "'THE MAOHAUS' 16 BanQiao Hutong Facade," *AntiStatics Architecture Design,* 2017.