CS230 Project: 2D Character Sketches to 3D Models

Andrew C. Yu
Department of Electrical Engineering
Stanford University
acyu@stanford.edu
Project Code: https://github.com/nmosfet/cs230-acyu-gfur

Abstract

3D modeling for video games or animation is time consuming and artists are expensive. To speed up content creation, we use a neural network autoencoder to convert hand-drawn, grayscale front and side 2D character sketches into a point cloud representation of a 3D model. The neural net is able to reconstruct training data, but fails to generalize to a novel dev set. This is largely due to a small training set (only 40 examples). These results nonetheless suggest that 3D point cloud reconstruction from a 2D image is possible. However, 3D point cloud representations are not scalable and fail to capture finer details, and as such are highly limited for actual applications.

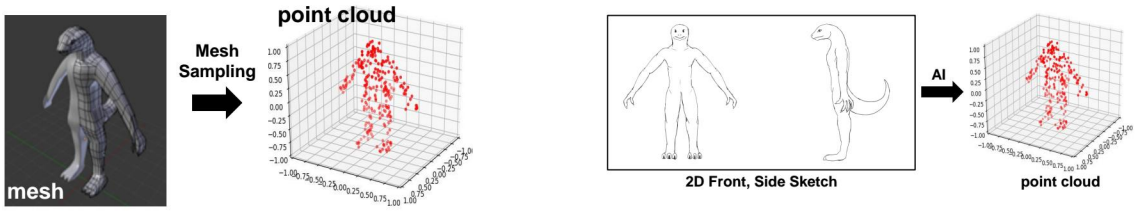
1 Introduction

We use a neural network to convert a hand-drawn, grayscale front and side 2D character sketch into a 3D model (**Fig. 1b**), in order to speed up asset generation for video games and animation. 3D modeling is time consuming and artists are expensive, so a method to generate 3D assets would help low-cost rapid prototyping.

Traditional 3D reconstruction from images typically involved photographs at multiple angles and assumptions about the camera's projection matrix [1]. But in the past five years there have been several works on 3D reconstruction from a single image. Buchanan et al. [2] used a non-neural-net approach that fits a skeleton into the outline of an image, but their approach does not work well on images with holes. Several neural net approaches now use deep convolutional networks to perform 3D reconstruction from single images [3; 4; 5; 6].

However, all prior neural net approaches only use a single view of a photograph input. So unlike prior work, this project (**1**) focuses on hand-drawn character sketches as inputs and (**2**) uses two images (front and side view) in the hope that two views will allow more generalizable 3D model features.

The neural network models and architectures used are primarily derived from Hinton et al. [7] and Fan et al. [5], with simplifications discussed in Section 3.



(a) (b)

Figure 1: (a) Mesh sampling to get 3D point cloud. (b) Project concept: hand drawn 2D character sketches as model input to generate 3D model output using a neural net.

2 Dataset and Features

2.1 2D Image Inputs

The 2D character sketches are 8 bit grayscale images. These are scaled to (128,128,1) matrices, where each entry is an 8 bit integer.

2.2 3D Point Cloud Sampling

The raw 3D models are meshes (vertices and faces), but it is difficult to do mesh loss and generation. Meshes are essentially graphs, and a graph loss function is difficult to define. Furthermore, it is difficult to ensure all meshes have the same number of vertices and faces.

Instead, for simplicity we represent the 3D output as a point cloud sampled from the mesh surface, depicted in **Fig. 1a** (using Pyntcloud library [11]). We take 256 sample points, so the 3D model format is a (256,3,1) matrix, where each entry is an (x,y,z) point. These points are normalized and zero-centered, to ensure similar spatial output between different models. First we subtract the mean independently along the x-, y-, and z-axes to zero-center. Then we divide by the max absolute value of all x, y, or z, confining all points $x, y, z \in [-1, 1]$.

2.3 Data Set for Project

The raw data used was 24 3D models and 2D front/side sketches. The majority of these models were taken from online sources [10], but some were hand-made. The 2D front/side sketches were all hand drawn for testing. The extremely small size of this data set is because: (1) there is no existing data set for this task and (2) I was a one man team.

The raw data was split into 20 training examples and 4 dev examples. The training data was augmented by vertically flipping the 2D images and 3D models (along z-axis), giving a total of 40 training examples.

3 Methods

3.1 Chamfer Distance Loss

We use the Chamfer distance L_{CD} as the loss function for two sets of points $S, \hat{S} \in \mathbb{R}^3$ [5]:

$$L_{CD}(S, \hat{S}) = \sum_{y \in S} \min_{\hat{y} \in \hat{S}} \{\|y - \hat{y}\|_2^2\} + \sum_{\hat{y} \in \hat{S}} \min_{y \in S} \{\|y - \hat{y}\|_2^2\} \tag{1}$$

This function sums the distance between each point in each set and its nearest neighbor in the other set. This ensures that the generated points are compared to their closest "equivalent" in the input set.

3.2 Autoencoder Architecture

The general neural net architecture is an autoencoder (**Fig. 2**), inherited from prior work by Fan et al. [5] and Dosovitskiy et al. [3]. The autoencoder begins with an encoding layer that detects features in the input image. These features feed into a decoding layer, which translates features into 3D points.

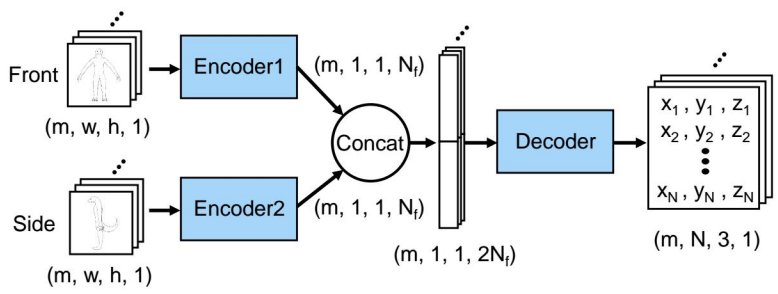


Figure 2: General autoencoder architecture overview, with two parallel encoder branches.

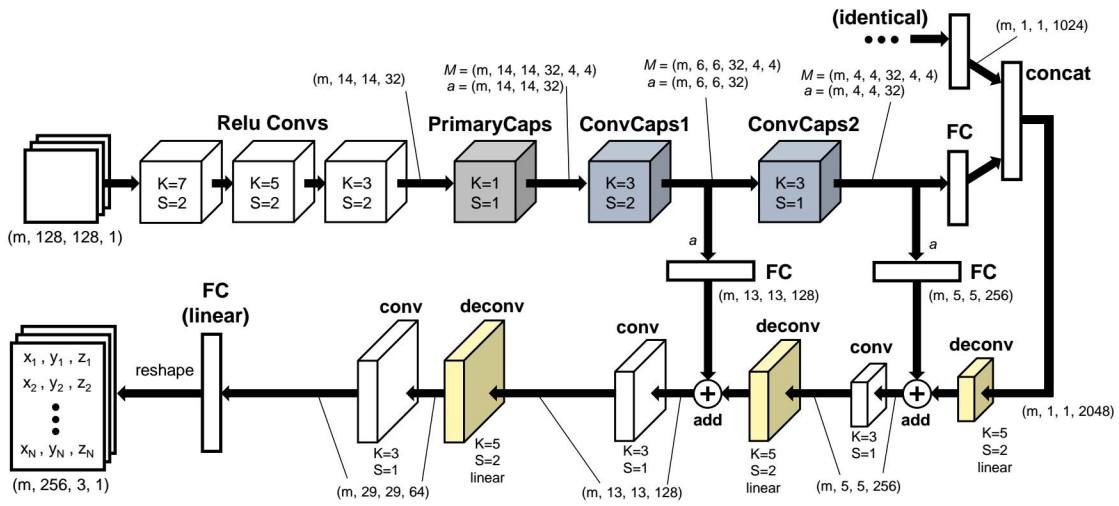


Figure 3: Encoder-decoder network architecture. In each layer, K is the convolution kernel size (K $\times$ K), and S is the stride. Only one encoder branch is depicted, but the other branch is identical and uses same weights.

In order to use a front and side view, our architecture uses two encoders which take in each view independently, then concatenates the detected features. These are fed into a single decoder. Each encoding layer is implemented using a Capsule Network [7]. The decoding layer is a typical upconvolution network that takes in features from different layers of the encoder [5]. **Fig. 3** depicts our detailed network implementation.

3.2.1 Encoder Capsule Network

Both encoders consist of Convolution (Conv) into Convolutional Capsule (ConvCap) layers. The intial 3 Conv layers with stride $S = 2$ reduce the input spatial dimension. A CapsNet is essentially a ConvNet with the following modifications:

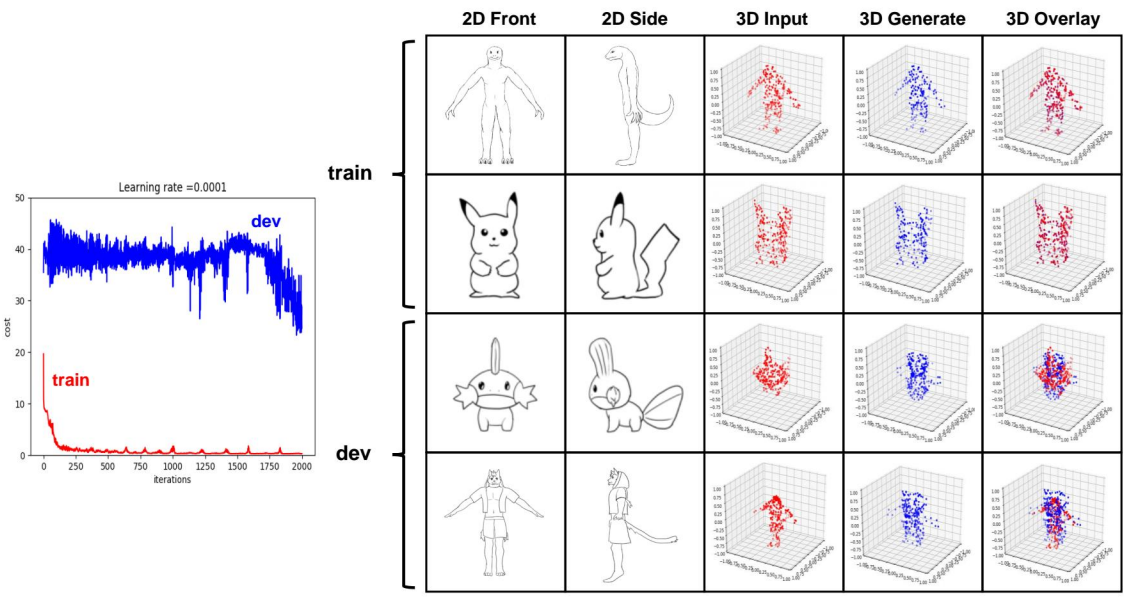
- Capsule layer output activations are tensors rather than scalars. Hinton et al. terms these tensor outputs as "capsules" [7]. In our case, each capsule layer output "capsule" is a feature activation a and a 4x4 pose matrix M that represents the feature's rotation.

- The outputs M and a go through a non-linearity from "dynamic routing" that tunes the probabilities that they are assigned to a weight in the next layer. Conceptually, this routing clusters "lower-level" features (i.e. an eyeball or a mouth) as parts of a "higher-level" feature (i.e. a head). If lower level features activate with high probability, the higher level feature will then activate. Hinton et al. term this dynamic routing as "EM Routing" because it uses an iterative expectation-maximization (EM) algorithm, which effectively fits Gaussian probabilities that the lower level features belong to a higher level feature [13]. For mathematical details, see [7; 9].

A CapsNet can supposedly generalize better from fewer training examples and be rotation invariant. Hence, theoretically the identical CapsNet encoder should determine that the front/side views belong to the same output object. However, the latest work on CapsNet are within the past five months [8; 7], so the benefits of CapsNet are still unclear. This project did not have enough training data to compare a CapsNet encoder versus a vanilla ConvNet encoder. The CapsNet was used mostly for fun.

Code implementations for the PrimaryCaps and ConvCaps layers were taken from a blog post by Hui [9].

Additionally, dropout layers are inserted before the fully connected (FC) layers from encoder output to decoder (with keep probabilities of 0.7 or 0.8). From testing, without these dropout layers, the generated points tend to cluster around a few input cloud points. Dropout before FC gives the crucial random "kick" that spreads the generated points more uniformly across the entire output spatial volume.



(a) (b)

Figure 4: (a) Cost function for train and dev set vs epochs. (b) Model generated point clouds at epoch = 1500, for typical train and dev example. Train examples are typically fitted closely, but not dev examples.

3.2.2 Decoder Upconvolution Network

The decoder consists of multiple deconvolution ("convolution transpose") layers with linear activation followed by a relu convolution layer. Deconv-conv layers perform the inverse of a convolution into pooling layer by increasing the output spatial dimension. The decoder ends with a linear FC layer to add scaling and shifting.

This decoder network is based on the "hourglass" decoder from Fan et al. [5], which also includes a parallel decoder path consisting of a single FC layer, which supposedly helps for "intricate" structures [5]. The outputs from the parallel deconv-conv and FC layers in the decoders are concatenated to give total generated output points. However, we found that this did not affect the output much, so we removed it.

Rather, the most important part of the decoder is the feedforward activations from earlier encoder layers. From testing, without these feedforward activations, the decoder output overfits to specific training examples. Adding the feedforward activations lets lower level features affect details in the output, reducing overfitting.

4 Results

This model was implemented in Tensorflow-slim for simplicity [12] and trained on an Amazon Web Services (AWS) EC2 p2.xlarge instance with a Tesla K80 GPU. A minibatch size of 4 was used due to GPU memory constraints. While according to Hinton et al. a CapsNet has fewer trainable parameters than an equivalent ConvNet [7], CapsNet uses significantly more GPU memory due to tensor copies for EM routing. A very low learning rate of 0.0001 was used, because higher values tended to cause the training cost to oscillate. The model was trained for 2000 epochs, with snapshots of generated point clouds every 100 epochs.

The cost function over training epochs is depicted in **Fig. 4a**. Examples of generated point cloud outputs compared to the input point clouds is depicted in **Fig. 4b**. The loss for the dev set in **Fig. 4a** is highly noisy and oscillatory, which is not surprising given that there are only 4 dev examples and 40 train examples. The generated point clouds can fit the train set pretty closely, but fail to fit the dev examples. The generated point clouds for the dev set are like averages across different training examples that are slightly similar to the dev input image. But the model is still overfitting to particular examples, for which there is not much that can be done beyond acquiring more training examples.

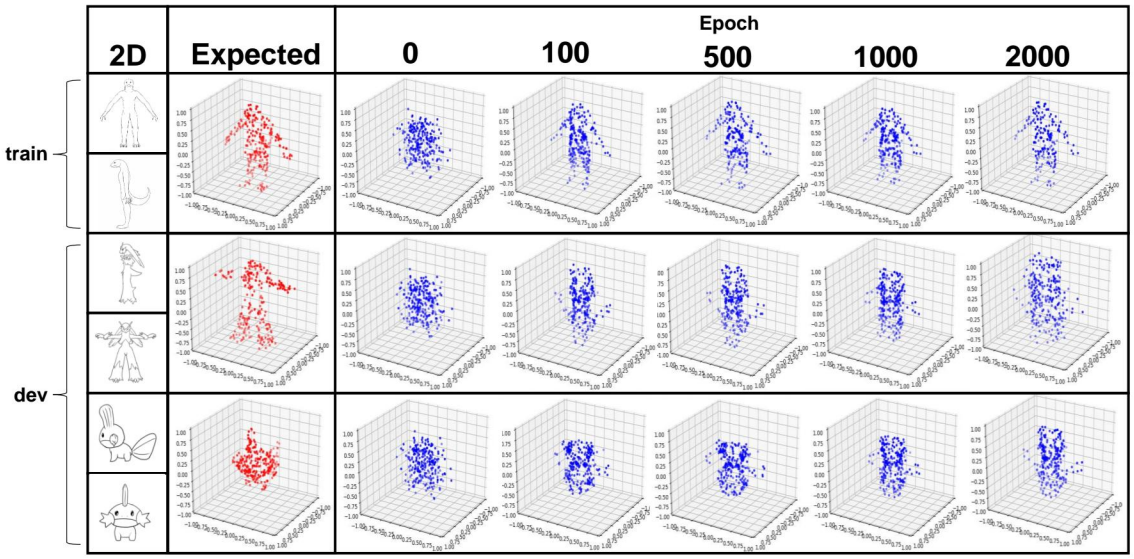


Figure 5: Snapshots of generated point cloud output for a train example and two dev examples during 2000 epochs. The output dev point cloud changes significantly over time, likely due to randomness from the dropout layers.

Fig. 5 examines how the generated point clouds evolve over epochs. All the generated clouds begin as a random blob of points. By epoch 100, the train example's generated cloud already acquires resembling its expected features. However, the dev examples are still fairly random. Over epochs, the generated dev clouds evolve randomly, likely due to the effects of dropout layers.

Unlike problems such as image classification, there are no good quantitative metrics to benchmark Bayesian or human error for creating a 3d mesh from 2d concept sketches. The visual eye can easily see if a model "looks right." But quantifying this as a percentage accuracy is difficult, as models can simply look "good enough." As such, quantitative analysis is limited here. The human can see that the generated dev point clouds have some features of the input 2D image, but clearly fail to approximate a desired model.

5 Conclusion and Future Work

We implemented an autoencoder network that can convert 2D grayscale front and side character sketches into a 3D point cloud representation of a 3D mesh. The autoencoder uses a capsule network to encode image features, followed by a deconvolutional network decoder to convert features into the 3D point cloud. These results are promising in that they can fit train examples pretty closely. However, due to the dearth of training examples (only 40), the model is unable to generate reasonable point clouds from novel images. The dev set point cloud outputs are more like averages of different train examples.

Aside from acquiring more training examples, this model could be improved by the following:

- **Adding more feed forward activations into decoder.** Fan et al. uses a more complicated recurrent network between the encoder and decoder [5].
- **Farthest point sampling and point ordering.** Output point clouds are unstructured, random data, which is difficult to generate with good spatial uniformity. Using a more uniform mesh sampling like farthest point sampling, and ordering the points (such as clockwise from top down), may improve point cloud generation.

However, this project exposes issues with 3D point cloud based representations. Point clouds scale poorly and cannot approximate fine details. As such, for long term, a mesh based approach is likely more useful, although more difficult to implement [6].

6 Contributions

I was the only team member. (Usage of "we" and "our" is just stylistic choice.)

7 Project Code

Project github repo: https://github.com/nmosfet/cs230-acyu-gfur

References

[1] https://en.wikipedia.org/wiki/3D_reconstruction_from_multiple_images

[2] P. Buchanan, R. Mukundan, and M. Doggett, "Automatic single-view character model reconstruction," *Proc. - Sketch-Based Interfaces Model. SBIM 2013 - Part Expressive* 2013, pp. 5–14, 2013.

[3] A. Dosovitskiy, J. T. Springenberg, M. Tatarchenko, and T. Brox, "Learning to Generate Chairs, Tables and Cars with Convolutional Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 692–705, 2017.

[4] C. B. Choy, D. Xu, J. Gwak, K. Chen, and S. Savarese, "3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction," vol. 1, pp. 1–17, *arXiv* 2016.

[5] H. Fan, H. Su, and L. Guibas, "A Point Set Generation Network for 3D Object Reconstruction from a Single Image," *arXiv* 2016.

[6] J. K. Pontes, et al., "Image2Mesh: A Learning Framework for Single Image 3D Reconstruction," *arXiv* 2017.

[7] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," *Iclr* 2018, no. 2011, pp. 1–12, 2018.

[8] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules," *Nips*, 2017.

[9] J. Hui, https://jhui.github.io/2017/11/14/Matrix-Capsules-with-EM-routing-Capsule-Network/, 2018

[10] https://www.yobi3d.com/

[11] https://github.com/daavoo/pyntcloud

[12] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

[13] https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm

A Appendix 1: Additional Generated 3D Point Clouds