Pixel2Mesh2*

*Note: Sub-titles are not captured in Xplore and should not be used

Peter Heile
Indiana University
peterjheile@gmail.com

***Abstract*—This paper covers a comprehensive documentation of the reimplementation of the original Pixel2Mesh framework [1], which aims to reconstruct detailed 3D mesh models from single RGB images. The primary objective is to develop a neural network capable of accurately inferring these 3D geometries from those single view images. The paper covers the setup, architectural design, experimental results, identified limitations, and implementation insights derived from this work.**

I. Introduction

Recreating meshes - the standard method for representing 3D objects - has long been a fundamental challenge in the computer vision field, especially from that of a single image. With this in mind however; there are countless different ways that this application could potentially be applied, including: robotics, helping machines understand hidden geometry; digital content creation; creating complete 3D structures for games or art; medical imaging, etc.

Traditional methods are typically reliant on volumetric data or point cloud representations which lack the detailed surface information needed for more precise prediction results; furthermore, they are often very computationally expensive - leading to overimploded resource usage and limitations. That being said, there have recently been introduced mesh based approaches that help solve these problems by creating more efficient and detailed reconstructions.

The Pixel2Mesh framework, is one of such methods. It presents and complete end-to-end solution that generates 3D mesh models from single RGB images. It works by progressively reforming an initial mesh using graph based convolution neural network (GCN), until an accurate object is created. This paper will go in depth into this method, including the challenges faced and the insights gained.

A. Motivations

The primary motivation for reimplementing Pixel2Mesh is to gain a deeper understanding of mesh-based 3D reconstruction techniques and to validate the reproducibility of the original model's results. By reconstructing the model from the ground up, I aim to identify potential areas for improvement, learn in depth the key features for 3D reconstruction, and develop a codebase that will integrate modern Python libraries into the workflow - allowing for future developers to integrate their efforts with less hassle.

Identify applicable funding agency here. If none, delete this.

Notably though, I am also fueled by a personal motivation due to my fascination with 3D objects and algorithms. A younger, much less experienced version of myself attempted a similar endevour with little-to no results, and now is a time for me to revisit and learn where I went wrong, and where I could have improved.

B. Background and Related Work

Early approaches to 3D reconstruction from single images predominantly utilized volumetric representations. While effective, these methods often suffer from high computational costs and limited resolution. Point cloud-based methods offer a more memory-efficient alternative but can lack surface connectivity information.

Pixel2Mesh introduces a novel approach by representing 3D shapes as meshes and employing GCNs to iteratively deform an initial ellipsoid mesh to match the target shape. This method leverages perceptual features extracted from the input image and iteratively refine the mesh over each stage. The integration of image features with mesh deformation allows for detailed and accurate reconstructions.

Subsequent works, such as Pixel2Mesh++, have extended this framework to handle multi-view inputs, further enhancing reconstruction quality. However, the original Pixel2Mesh remains a foundational model in single-view 3D mesh reconstruction, making its reimplementation a valuable journey in advancing my breadth of knowledge in this field.

C. Implementation Overview

My reimplementation of Pixel2Mesh is developed using PyTorch, aiming to replicate the original model's architecture and training procedures. The implementation consists of two primary components:

Image Feature Extraction: A Convolutional Neural network (CNN) is used to extract the perceptual features from the input RGB image. Despite the original paper using a VGG-16 backbone, I opt for a more "modern" ResNet-18 backbone.

Mesh Deformation Network: The initial mesh, an ellipsoid with a fixed topology, is progressively deformed through a series of graph convolution layers. Each deformation stage incorporates features from the image to guide the mesh refinement process.

Key aspects of the implementation include the integration of perceptual feature pooling, the design of mesh-specific loss functions (such as Chamfer distance and Laplacian regularization), and the adoption of a progressive deformation method. Throughout the reimplementation, emphasis is placed on modularity and clarity to facilitate future research and development, and many design choices I made were with this in mind.

II. Methods

This section outlines the approach taken to reimplement the Pixel2Mesh framework, which reconstructs 3D mesh models from single RGB images. The reimplementation was conducted using PyTorch (applying gpu acceleration where possible), adhering closely to the architecture and methodologies described in the original Pixel2Mesh paper, though admittedly, taking several creative liberties to reduce compute time and ensure simplicity for future developers, or others interested. The following subsections detail the dataset preparation, network architecture, training procedures, and evaluation metrics employed in this study.

A. Dataset

For this project, I use the *ShapeNetCore* dataset, from which access can be requested and granted via Huggin Face [2]. The Shapenet dataset includes approximately 55 classes and 51,000 unique 3d models stored as .obj files. From this I downloaded a few different class, though namely, and the one used for my application, I downloaded the airplane class (id: 02691156). In addition to this, there was an original dataset file used for the original P2M model that I also downloaded [**?**]. It provided 5 different perspective images and the corresponding .dat file for each perspective (.dat files included the vertex [x,y,z] positions and vertex normal as 6 values in a line for each row).

To ensure I maintained all of the necessary information, (including additional data for visualizations), I decided to pool these dataset together, adding object files to the respective instances in the P2M dataset. Once finished, I was left with a cleaned_dataset directory that I used as my version of the dataset used for P2M.

To complement this newly made cleaned_dataset, I implemented my own custom subclass from the pytorch datasets library. This allowed me to use this directory to load all of the respective instance and class data into a dataset object. Within this object, I stored: image_data, ground_truth points, ground_normals (vertex), K, verts, faces, and meta informations such as class_id and instance_id. Note that the verts and faces information comes from the Shapenet .obj file, and is there solely for visualization purposes. Grount truth information used for training was pulled from each resepective .dat file for each input image. This subclassing further allowed me to maintain the torch pipeline and functions, meaning that additional functions such as batching, parallel data loading, subscripting, etc were all inheretly included in this dataset class. Below provides a summary of the information printed from this dataset.

B. Preprocessing

As far as preprocessing goes, most of the hard work had already been done before downloading the dataset. For each input image, the respective 3D vertices and vertex normals were already calculated and normalized.

However, for each input image, I did have to normalize the respective features and resize to the appropriate dimensions required for inputting into the ResNet input. This was all done within my custom dataset class, ensured the maintained modularity and hiding a lot of the complexity under the table for future observers.

C. Network Architecture

There are two core components of the model architecture used in this project. The first being the ResNet-18 backbone and the second being three (3) Mesh Deformation Blocks.

1) ResNet Backbone: The first core component of the Pixel2Mesh architecture is a Feature Extractor. This feature extractor coagulates relevant information and data from each image. For my use case, I opted on changing the feature extractor from the original paper to the ResNet-18. This backbone I loaded **pretrained** and wrapped it into a separate PyTorch model. When inputting an image, the features from each of 4 blocks at different layers within the ResNet were collected and stored as layer variables.

Input images were required to be of size (1, 224, 224, 3) with the first indices being the batch size, the next two the dimensions, with third the number of channels.

Again, I maintained a focus on keeping a simplicit and modern model for abstraction and future implementations. This wrapper I made includes all pytorch model features, such as gpu loading, backpropogation loading, etc. This allows for me to already have many of the nccessary function build into the model

2) Graph Convolutional Network: The second core component of the P2M is three different mesh deformation blocks. Each of these blocks consists of Graph conolutional layers. The connections between these layers was determined by the face information of the initial input mesh. The output of these Graph convolutional layers is a (n, 3) size, representing the [x,y,z] position change of each vertices from the initial mesh.

In addition to having graph convolution layers, each Mesh deformation block included a feature pooling segment at the beginning. This application used the features extracted from respective layers of the feature extractor. The 3D input mesh is then projected onto this feature plane to find the features that correspond to each vertex. Note that I made the creative choice to sample the features down with the intent of trying to provide less compute time, while hoping that enough features would still be included to annotate vertices with relevant information.

At the last segment of each deformation block was a unsampling section. In this unsampling, the outputted mesh from the model would have vertices added, splitting each face into essentially 4 new faces. This would quadruple the number of vertices/faces and allow for the next stage to have better refinement. This portion was kept for each of the mesh deformation blocks, except for the last.

Again, I attempted to keep as close to the pytorch library as possible thorughout this process. This again helps to maintain modularity. That being said, due to my use of a third part library for the mesh unsampling, the model is unable to be batching, meaning it will have to train on sequential inputs.

The below chart provides the relative feature counts, sampled feature counts, mesh vertex counts, and ResNet layer used to project the mesh to those features.

Fig. 1. This is a sample image.

D. *Model Training*

This section gives an overview on the loss functions used for training the model and the method used to training (such as train/test split).

1) Loss Functions: There were four (4) loss functions used to train this model. Below listed is each loss function, with the corresponding equation for each.

- **Chamfer Distance Loss:**
$$\mathcal{L}_{\text{Chamfer}}(P,Q) = \frac{1}{|P|}\sum_{p\in P}\min_{q\in Q}\|p-q\|_2^2 + \frac{1}{|Q|}\sum_{q\in Q}\min_{p\in P}\|q-p\|_2^2$$
- **Edge Length Regularization Loss:**
$$\mathcal{L}_{\text{edge}} = \sum_{(i,j)\in\mathcal{E}} \left(\|v_i - v_j\|_2 - \ell_{ij}\right)^2$$
where $\mathcal{E}$ is the set of edges, v_i and v_j are vertex positions, and ℓ_{ij} is the target edge length.
- **Laplacian Smoothness Loss:**
$$\mathcal{L}_{\text{lap}} = \sum_i \left\| v_i - \frac{1}{|\mathcal{N}(i)|}\sum_{j\in\mathcal{N}(i)} v_j \right\|_2^2$$
where $\mathcal{N}(i)$ denotes the set of neighboring vertices of vertex i.
- **Normal Consistency Loss:**
$$\mathcal{L}_{\text{normal}} = \sum_{(f_i,f_j)\in\mathcal{A}} \left(1 - \mathbf{n}_{f_i}\cdot\mathbf{n}_{f_j}\right)$$
where $\mathcal{A}$ is the set of adjacent face pairs, and $\mathbf{n}_{f_i}$ and $\mathbf{n}_{f_j}$ are the unit normal vectors of faces f_i and f_j, respectively.

Each of these loss functions have a different purpose. The chamfer loss is the main driver of the model. It helps to transform the vertex positions of the input mesh to align closer with vertices positions of the ground truth mesh by finding the average distance between each points of the two meshes. The edge length loss helps to encourage more uniform edge lengths by penalizing deviations from target edges. The laplacian smoothness loss help to make the mesh look "smoother" by minimizing the differences between each vertices by using the average positions of its neighbors. The Normal loss penalizes the differences in orientation between adjacent faces, also helping to promote a smoother mesh surface.

Each of these loss functions is assigned a weight, from which they are summed together to make the total loss. These weights and "total loss" is defined for each of the three mesh deformation blocks. Then each of these "total losses" from each block is again assigned a weight, and summed to have the loss of the entire model. This allows us to hopefully, not only choose which blocks focus on minimizing vertex differences, mesh smoothness, etc, but also define which block specializes in which area. Below shows the assignments of one such of these loss weight definitions.

There were several training regimes that I used in an attempt to tune this model, such as 5fold cross validation; however, due to

III. RESULTS

Due to the complexity of analyzing mesh structures, a lot of my analyses I left to the "visual guide" and loss analyses. For example, with preliminary testing my model would have trouble learning any of the loss functions at all, with each loss chart portraying erratic behavior. From this, I would manually tweak weight values, sampling counts, and then try again. From these charts, I could relatively tell how well my model was learning each the shape of the airplane class, and I would make weight choices based on it. For example, when decreasing the weight position of the entire first block, the chamfer loss of the first block, the chamfer distance between the ground truth vertices and the predicted vertices actually decreased. Note that this does not mean that a more accurate mesh was outputted though, as the edges and smoothness could still change for the worse.

The second method I mainly used was my own opticaled. After training each model for a period of time, I would notice a low loss value for the model. From which, I would load the corresponding model checkpoint and make a prediction on a test image. From there, I could load the mesh after each stage and note how the model mesh refinement progressed over time. Although the second and third layers would often undergo drastic mesh deformation, one artifact that I could not solve was that the mesh would deform little to none after the fist stage. However, according to the original Pixel2Mesh paper, the "airplane" shape should already be somewhat visible at this stage.

The third method I used was at the hope of giving me some basic guideline on how well my model was performing, due to myself having trouble udnerstanding how "well" my model was actually performing. Though this is by no means an accurate or encompasing test on accuracy, and was more for my sole relative understanding, I would take the average pointcloud difference of around 100 differnt input images, and use a prebuild numpy library to normalize the result between 0-1. Below shows the accuracy result after one such run.

IV. DISCUSSION

Overall, to be honest, the quantitative results of my P2M repoductionw were lacking, and a little frustrating. However, in hindsight, there are several core factors that I can link these two. The first of which, and by far the most impactful, is my use of sampling the ground truth vertices to reduce the pointcloud size. Due to past attempts at similar projects, I was adversize to training a heavy model on my computer. Because of this, I reduced the samples to 1024 vertices knowing this would drastically reduce the training time and calculations of the loss functions. In secondary testing (after my conclusive results), I decided I would try to double this sampling to 2048. Immediately, with no other changes, my training performed noticeably better with my model achieving lower losses in roughly the same time span. Note that I trained on average each model attempt for appx 2-3 hours. Even then, from this I can infer that if I even removed the grount truth sampling altogether, and left each isntance with its roughly 200,000 vertices, results would most likely be drastically better.

The second contributor to the negative results in my opinion was the lack of a structured weight selection. Due to the complexity of the weight functions (4 weights per block times 3 blocks), I failed to come up with a good solution to finding the optimal weights per weight and per block. Furthermore, trying every combination was impractical considering the 2-3 hour train time per model even with the reduced sampling to 1024 vertices.

With all of this being said and done, I am still very happy with my results. I have created a very intuitive Pixel2Mesh model that does in fact attempt to solve the problem (there are no core architectural issues that I am aware of). At this point, it seems to me that the biggest factor to allowing this model to perform at the original P2M model level is through days of computing time, higher samples, and better wieght selection. All of which were done in the original P2M model (3 days on a GPU training).

In addition to this, I also learned in depth the entirety of the core principles of this P2M implementation, along with amny of teh core principles of 3D asset generation as a whole. I feel completely confident in my udnerstanding of the subject and my ability to apply it in future projects, and witht eh future developement of this project.

V. Conclusion

My project aimed at reimplementing the Pixel2Mesh framework from a single RGB image. It had two main goals, the frist of which was to replicate the results of the original paper, and the second to help give myself a comprehensive understanding of 3D generation as a whole. While the original implementation of the model deomstrated extremely impressive results, generating accurate 3D meshed, my reimplementation fell far short results-wise. This discrempancy very much highlights the coomplexity involved in reproducing advanced deep learning models and also helpo underscore the importance ofr thorough documentation and understanding of the underlying methodologies.

Despite these challendes encountered, my reimplementation did provide valuable insights into the architecture and functioning of the Pixel2Mesh model, especially for my own personal learning. Further insights were that using different backbones and componenets (such as my ResNet-18 backbone), are compeltely viable and transmutable in the architecture - though the impace on perfomance is left to be collected and analyzed.

The potential impact of successful 3D mesh reconstruction from single images is significant, with applications spanning augmented reality, robotics, and digital content creation and more. Advancements in this area can lead to more immersive virtual experiences, improved object recognition in autonomous systems, and streamlined workflows in creative industries.

Future work should focus on identifying and addressing the factors that led to the performance gap in this reimplementation. This may involve exploring alternative network architectures, refining training protocols, or leveraging more diverse and comprehensive datasets. Collaborative efforts and open-source contributions can also play a pivotal role in enhancing the reproducibility and robustness of such complex models, as I also did have the limitation of being the sole contributor to this project.

In conclusion, while this reimplementation did not replicate the success of the original Pixel2Mesh model, it served as a valuable learning experience and a stepping stone toward further exploration and improvement in the field of 3D mesh reconstruction.

Acknowledgment

References

[1] @inproceedingswang2018pixel2mesh, title = Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images, author = Nanyang Wang and Yinda Zhang and Zhuwen Li and Yanwei Fu and Wei Liu and Yu-Gang Jiang, booktitle = Proceedings of the European Conference on Computer Vision (ECCV), pages = 52–67, year = 2018, publisher = Springer, doi = 10.1007/978-3-030-01252-6$_4, url = https://arxiv.org/abs/$1804.01654

[2] @techreportshapenet2015, title = ShapeNet: An Information-Rich 3D Model Repository, author = Chang, Angel X. and Funkhouser, Thomas and Guibas, Leonidas and Hanrahan, Pat and Huang, Qixing and Li, Zimo and Savarese, Silvio and Savva, Manolis and Song, Shuran and Su, Hao and Xiao, Jianxiong and Yi, Li and Yu, Fisher, number = arXiv:1512.03012 [cs.GR], institution = Stanford University — Princeton University — Toyota Technological Institute at Chicago, year = 2015

[3] @inProceedingswang2018pixel2mesh, title=Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images, author=Nanyang Wang and Yinda Zhang and Zhuwen Li and Yanwei Fu and Wei Liu and Yu-Gang Jiang, booktitle=ECCV, year=2018