Deep Learning for Shape Reconstruction: Converting 2D Images into 3D Voxel Grids Using ShapeNet Data

Ernie Sumoso
Artificial Intelligence & Machine Learning
Lambton College
Toronto, Canada

Abstract— **The generation of 3D objects from 2D images presents a significant challenge in computer vision, with applications ranging from augmented and virtual reality to medical imaging and automated quality control. Traditional methods for 3D reconstruction rely on computationally intensive algorithms and pre-existing 3D data. This project explores the use of Convolutional Neural Networks (CNNs) to train a model capable of accurately reconstructing voxel grids from 2D image inputs. We dealt with over 480,000 images, approximately 48,000 OBJ files from the ShapeNet Core dataset to train 50 CNN models.**

Our approach involves generating 2D images from 3D models in the ShapeNet dataset using an in-house modified version of the Stanford ShapeNet Renderer. These images, combined with the 3D voxelized representations, are preprocessed into formats suitable for CNN training. By testing multiple model architectures, we are able to provide an in-depth understanding of CNN mechanisms for learning spatial relationships and patterns from images. The models are then evaluated using metrics like Intersection Over Unit (IoU), Accuracy in voxel occupancy prediction, and visual comparisons. The results suggest detailed 3D voxel grids, contributing to advancements in spatial data research and enabling faster and more accurate object recognition.

This work sets a standard for working with 2D and 3D data by experimenting with various models depending on the shapes and spatial data properties. Applications range from e-commerce, healthcare, and even manufacturing by providing an innovative approach for 3D object generation and understanding. Through this project, we developed expertise in deep learning, data processing, and AI-driven solutions for complex real-world problems. The final product includes an intuitive interface that allows users to upload 2D images and receive corresponding 3D object reconstructions, making the technology initially accessible and practical for testing and real-world applications.

Keywords — Deep Learning, Convolutional Neural Networks, 3D Shape Reconstruction, 3D Object Generation, Image processing, Image based training, ShapeNet, Voxel Grids Generation, Image data processing, OBJ files, 3D data processing, Machine Learning, Artificial Intelligence

I. Introduction

ShapeNet is an extensive dataset of 3D shapes categorized into distinct classes, commonly used in computer vision and machine learning for 3D object recognition tasks. With over thousands of unique 3D models, it includes objects such as airplanes, cars, and tables, stored in OBJ file format, which captures detailed vertices and faces. The dataset also provides annotations and category-specific labels, offering additional flexibility for segmentation and enabling specialized applications.

Fig 1. ShapeNet Core objects example

The original ShapeNet dataset has different versions such as ShapeNet Core, ShapeNetSem, ShapeNet Parts. However, we chose ShapeNet Core over other versions due to its balance of size, quality, and usability for our specific project needs. ShapeNet Core offers over 50,000 high-quality 3D models, organized into 55 categories, which provides a comprehensive dataset for training our CNN. Unlike ShapeNetSem, which is much larger but includes noisier data, ShapeNet Core maintains a higher level of consistency in its annotations, ensuring more reliable results during training. Additionally, its focus on widely recognized object categories aligns closely with our goal of generating 3D voxel grids from 2D images.

By training a CNN from scratch using the data, our goal is to develop a model capable of generating voxel grids, translating 2D images into 3D representations. Furthermore, constructing the model architecture from the ground up enhances our understanding of CNN mechanisms, providing valuable insights into how neural networks learn complex patterns.

The data details are provided in the following table for a better understanding of the initial data.

Table 1: ShapeNet Dataset Description

Field	Value
3D models count	48,958
3D models format	Object File (OBJ)

Categories (labels)	50
Dataset Version	ShapeNet Core

II. Data Processing

A. *3D shapes (OBJ) into Voxels into Numpy arrays (NPY)*

Our first step in data processing is converting the entire dataset into a more structured format, suitable for model training. We chose Numpy arrays due to their flexibility and ease of use when working with large amounts of data. First, we decided to implement a function to identify all OBJ files recursively given an initial path. So, given the main dataset root path we were able to save the path for all 48,000 OBJ files. The following figures show the distribution of categories (top 10, and all categories).

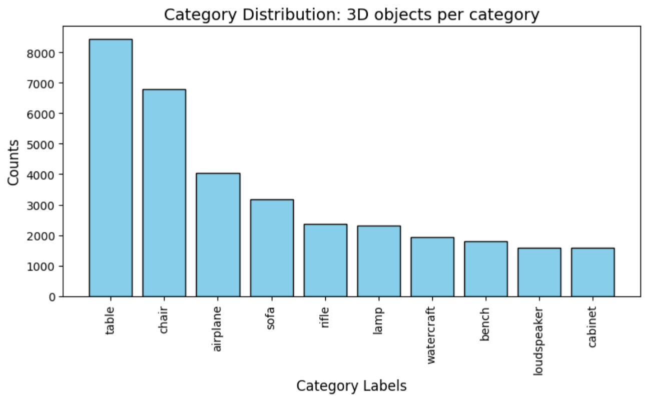


Fig 2. ShapeNet Core: Top 10 categories with the most 3D objects

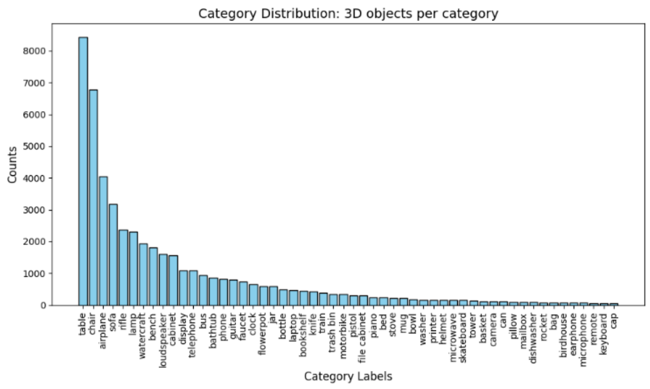
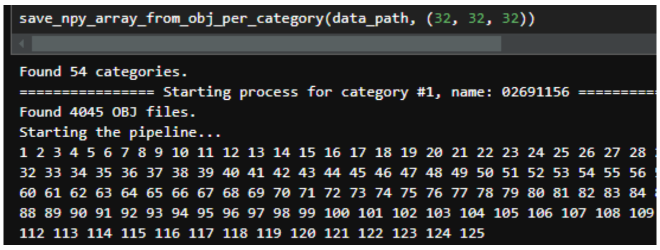


Fig 3. ShapeNet Core: Category Distribution, 3D objects per category

To convert the 3D shapes in OBJ format into voxelized representations and store them as structured NumPy arrays (NPY), we implemented a dedicated pipeline. The process began with reading the list of OBJ files identified during the initial recursive search. Each file represents a 3D mesh, which was loaded using the trimesh library.
Once loaded, the mesh is voxelized, transforming the 3D object into a discrete voxel grid representation. To ensure consistency and compatibility with our model, we specified a uniform voxel resolution and reshaped each voxel grid to the desired format, appending a single-channel dimension. This ensured the resulting arrays had a consistent shape of (32, 32, 32, 1) initially.

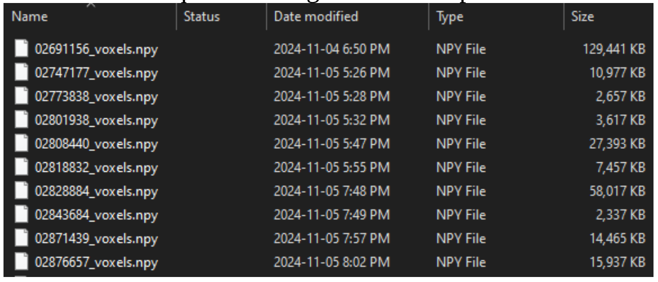


```
save_npy_array_from_obj_per_category(data_path, (32, 32, 32))

Found 54 categories.
================ Starting process for category #1, name: 02691156 ==========
Found 4045 OBJ files.
Starting the pipeline...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
112 113 114 115 116 117 118 119 120 121 122 123 124 125
```

Fig 4. Running pipeline to convert 3D object files into Numpy arrays

This final structured format is well-suited for efficient storage and direct use in training our convolutional neural network (CNN). Optionally, we saved the resulting array as an NPY file to facilitate quick loading for future experiments.



Name	Status	Date modified	Type	Size
02691156_voxels.npy		2024-11-04 6:50 PM	NPY File	129,441 KB
02747177_voxels.npy		2024-11-05 5:26 PM	NPY File	10,977 KB
02773838_voxels.npy		2024-11-05 5:28 PM	NPY File	2,657 KB
02801938_voxels.npy		2024-11-05 5:32 PM	NPY File	3,617 KB
02808440_voxels.npy		2024-11-05 5:47 PM	NPY File	27,393 KB
02818832_voxels.npy		2024-11-05 5:55 PM	NPY File	7,457 KB
02828884_voxels.npy		2024-11-05 7:48 PM	NPY File	58,017 KB
02843684_voxels.npy		2024-11-05 7:49 PM	NPY File	2,337 KB
02871439_voxels.npy		2024-11-05 7:57 PM	NPY File	14,465 KB
02876657_voxels.npy		2024-11-05 8:02 PM	NPY File	15,937 KB

Fig 5. Sample stored voxels in Numpy arrays as NPY files (1 per category)

B. *Rendering Images (PNG) for training*

To generate 2D images from the 3D models in OBJ format, we utilized the Stanford ShapeNet Renderer (SSR) tool. We edited this software developed in Python to suit our project needs and data without affecting the main image rendering that relies on open-source Blender 2.9.0 *[4]*. The SSR tool allowed us to specify various parameters, including the number of views and camera angle. Other variables like the image type (e.g., albedo, normal, or depth) were pragmatically removed to save resources and time during rendering. For each 3D object, we rendered 10 images from different perspectives, resulting in approximately 500,000 images in total.

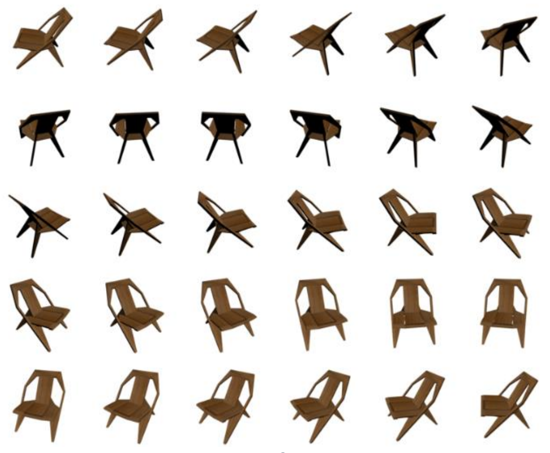
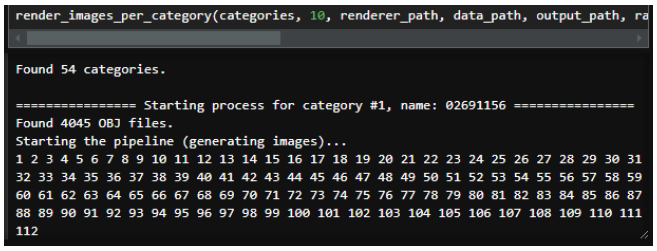


Fig 6: Example rendered images from a single 3D object, generated by the ShapeNet Renderer Standford tool

We implemented a custom pipeline to automate this process. For each category, the rendering script was invoked through Blender *[4]*, ensuring the images were generated efficiently and stored in a structured directory format.



```
render_images_per_category(categories, 10, renderer_path, data_path, output_path, ra

Found 54 categories.

=============== Starting process for category #1, name: 02691156 ===============
Found 4045 OBJ files.
Starting the pipeline (generating images)...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
112
```

Fig 7. Running pipeline to render images from 3D objects by using an in-house edited version of the Stanford ShapeNet Renderer tool

Due to the significant computational resources required for rendering such a large dataset, we distributed the workload across multiple machines, and even acquired new hardware to boost our processing times. This collaborative approach helped us overcome hardware limitations and complete the task in a reasonable timeframe. The resulting dataset of 2D images forms a vital part of our project, enabling the training of our convolutional neural network (CNN) for 3D shape prediction. With the images and 3D shapes datasets prepared, we can proceed to the next phase of pre-processing.

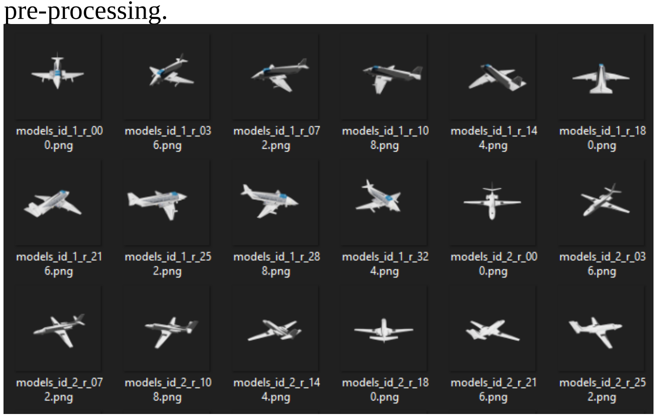
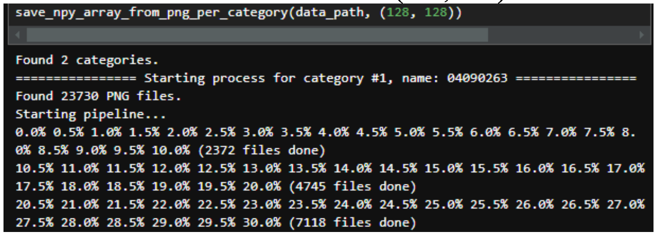


Fig 8. Example results of our rendering image pipeline

C. *Images (PNG) into Numpy arrays (NPY)*

To convert the rendered images into NumPy arrays, we implemented a processing pipeline suitable to the dataset structure. The process involved reading the PNG files, resizing them to a consistent shape, and storing the resulting data as structured NumPy arrays.

We began by identifying all PNG files in the dataset directory using a recursive search. This ensured that all rendered images across categories were included in the pipeline. Each PNG file was opened using the **Pillow** library and converted to grayscale to reduce computational overhead while preserving essential information. The grayscale images were resized to a fixed resolution of (128, 128).

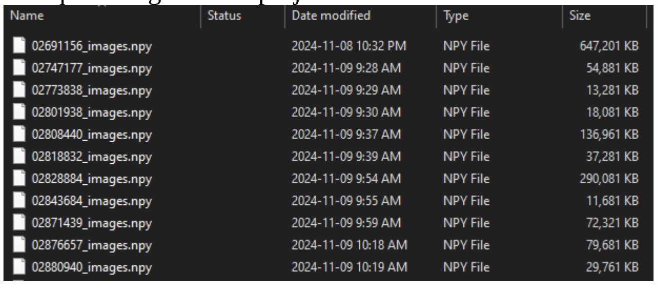


```
save_npy_array_from_png_per_category(data_path, (128, 128))

Found 2 categories.
=============== Starting process for category #1, name: 04090263 ===============
Found 23730 PNG files.
Starting pipeline...
0.0% 0.5% 1.0% 1.5% 2.0% 2.5% 3.0% 3.5% 4.0% 4.5% 5.0% 5.5% 6.0% 6.5% 7.0% 7.5% 8.
0% 8.5% 9.0% 9.5% 10.0% (2372 files done)
10.5% 11.0% 11.5% 12.0% 12.5% 13.0% 13.5% 14.0% 14.5% 15.0% 15.5% 16.0% 16.5% 17.0%
17.5% 18.0% 18.5% 19.0% 19.5% 20.0% (4745 files done)
20.5% 21.0% 21.5% 22.0% 22.5% 23.0% 23.5% 24.0% 24.5% 25.0% 25.5% 26.0% 26.5% 27.0%
27.5% 28.0% 28.5% 29.0% 29.5% 30.0% (7118 files done)
```

Fig 9. Running pipeline to save images as Numpy arrays (NPY)

After processing all images within a category, the arrays were stacked into different 3D Numpy arrays (1 per category) of shape (#instances, 128, 128). This structured format optimized the dataset for efficient storage and direct use in training. The processed data for each category was saved as an NPY file, with filenames corresponding to the category names. This allowed for easy retrieval and reuse of the data in subsequent stages of the project.



```
Name                   Status   Date modified          Type       Size
02691156_images.npy             2024-11-08 10:32 PM    NPY File   647,201 KB
02747177_images.npy             2024-11-09 9:28 AM     NPY File    54,881 KB
02773838_images.npy             2024-11-09 9:29 AM     NPY File    13,281 KB
02801938_images.npy             2024-11-09 9:30 AM     NPY File    18,081 KB
02808440_images.npy             2024-11-09 9:37 AM     NPY File   136,961 KB
02818832_images.npy             2024-11-09 9:39 AM     NPY File    37,281 KB
02828884_images.npy             2024-11-09 9:54 AM     NPY File   290,081 KB
02843684_images.npy             2024-11-09 9:55 AM     NPY File    11,681 KB
02871439_images.npy             2024-11-09 9:59 AM     NPY File    72,321 KB
02876657_images.npy             2024-11-09 10:18 AM    NPY File    79,681 KB
02880940_images.npy             2024-11-09 10:19 AM    NPY File    29,761 KB
```

Fig 10. Stored images in Numpy arrays as NPY files

D. *Overall Data Pre-processing Results*

Through our data preprocessing pipeline, we successfully transformed the ShapeNet dataset into a structured format optimized for deep learning. Approximately 50,000 OBJ files were converted into voxel grids, resulting in a comprehensive 3D dataset with a uniform shape of (32,32,32,1). Simultaneously, we generated over 500,000 images from these models and processed them into grayscale arrays with a consistent resolution of (128,128,1). This dataset of voxel grids and 2D images is now ready for efficient training and testing of our convolutional neural network models.

Table 2: Data Processing results description

Field	Value
# Images rendered	489,580
# OBJ files converted into voxel grids	48,958
# NPY files storing all voxels per category	50
# NPY files storing all images per category	50
Approximate computational hours exhausted	150
Libraries used for pre-processing	Numpy, Trimesh, PIL, glob, os

III. ADDRESSING CLASS IMBALANCE

To address class imbalance in our dataset, we segmented the data categories into three subsets based on the size of the data, allowing us to adapt the complexity of our CNN architectures to the available data. For the largest datasets, we employed a high-capacity, complex CNN architecture that had been previously tested and validated as our base model. For smaller datasets, we designed a low-capacity CNNs to prevent overfitting and optimize training efficiency. This approach ensures that the model complexity aligns with the dataset size, enabling effective learning across all categories while mitigating the challenges of imbalance.

In the following figure we visualize how we decided to divide our dataset categories based on number of instances. For high-capacity models we are going to use 6 categories that have large amounts of data:

Table 3: Categories used for high-capacity models

Category	**#Instances**
‘table’	8436
‘chair’	6778
‘airplane’	4045
‘sofa’	3173
‘rifle’	2373
‘lamp’	2318

Table 4: Categories used for low-capacity models

Category	**#Instances**
'watercraft'	1939
'bench'	1813
'loudspeaker'	1597
'cabinet'	1571
'display'	1093
'telephone'	1089
'bus'	939
'bathtub'	856
'phone'	831
'guitar'	797
'faucet'	744
'clock'	651

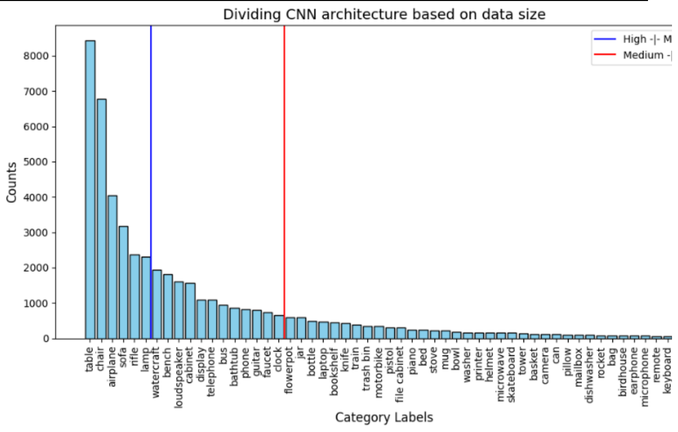


Fig 11. Dataset segmenting to address class imbalancing

Finally, on the final experiment, we also consider using the entire dataset (approximately 50’000 3D objects) to train a complex model. Let’s continue by engineering our CNNs.

IV. High-Capacity CNNs: Developing the CNN Architecture, Training and Testing

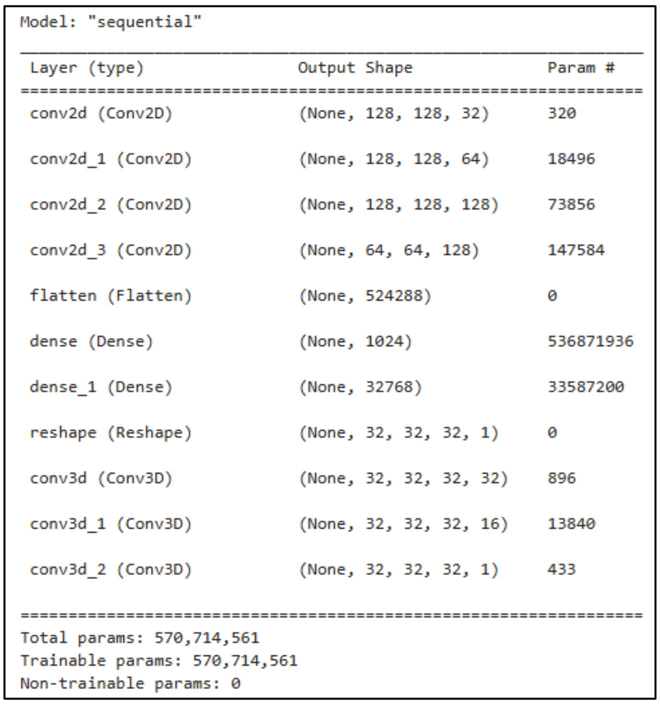
Once the data pre-processing is finished. We started engineering our CNN architecture. This was a trial-error approach with multiple architectures and data parameters to test. For this task we worked with Keras Sequential model and layers.

We built a convolutional neural network with the use of the following layers:

- Start by adding **2D Convolutional layers**.
- Then we **flatten** our output and add **dense layers** before reshaping into a 3D space.
- Finally, we add **3D Convolutional layers** to enhance the 3D spatial results

As we are working with large amounts of data (thousands of images and 3D objects), we had to come up with a high-complexity model architecture. Our final model presented the following layers and parameters. Overall, our model architecture had:

- 4 2D convolutional layers
- 1 flattened layer
- 2 dense layers
- 1 reshape layer
- 3 3D convolutional layers
- **570 million parameters** in total



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 128, 128, 32)      320

 conv2d_1 (Conv2D)           (None, 128, 128, 64)      18496

 conv2d_2 (Conv2D)           (None, 128, 128, 128)     73856

 conv2d_3 (Conv2D)           (None, 64, 64, 128)       147584

 flatten (Flatten)           (None, 524288)            0

 dense (Dense)               (None, 1024)              536871936

 dense_1 (Dense)             (None, 32768)             33587200

 reshape (Reshape)           (None, 32, 32, 32, 1)     0

 conv3d (Conv3D)             (None, 32, 32, 32, 32)    896

 conv3d_1 (Conv3D)           (None, 32, 32, 32, 16)    13840

 conv3d_2 (Conv3D)           (None, 32, 32, 32, 1)     433

=================================================================
Total params: 570,714,561
Trainable params: 570,714,561
Non-trainable params: 0
```

Fig 12. High-capacity CNN architecture layers

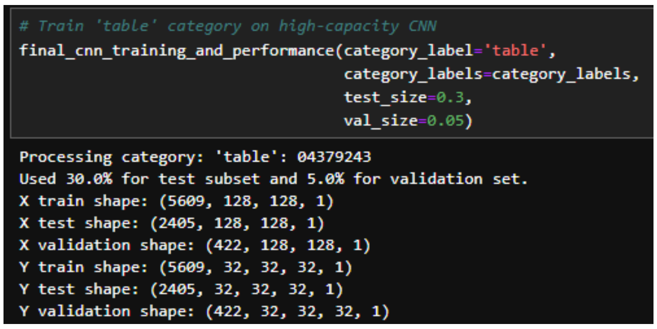
A. *Data Splitting Strategy*

To effectively train and evaluate our models, we implemented customized functions for data splitting. The data was first prepared by aligning 2D images with their corresponding 3D voxel representations based on the required number of images per 3D object.

We then split the dataset into training, testing, and validation subsets using predefined proportions. For this proportions we used to following percentages:

- **Testing subset: 30%** allocated
- **Validation subset** (unseen data): **5%** allocated
- **Training subset:** rest of the dataset

Additionally, an optional parameter allowed us to work with a specified percentage of the data for training and testing, enabling flexibility in data usage. This approach ensured balanced and representative splits across subsets, facilitating reliable model training and evaluation.



```
# Train 'table' category on high-capacity CNN
final_cnn_training_and_performance(category_label='table',
                                   category_labels=category_labels,
                                   test_size=0.3,
                                   val_size=0.05)
```

```
Processing category: 'table': 04379243
Used 30.0% for test subset and 5.0% for validation set.
X train shape: (5609, 128, 128, 1)
X test shape: (2405, 128, 128, 1)
X validation shape: (422, 128, 128, 1)
Y train shape: (5609, 32, 32, 32, 1)
Y test shape: (2405, 32, 32, 32, 1)
Y validation shape: (422, 32, 32, 32, 1)
```

Fig 13. *Dataset splitting results for category 'table'*

B. *Models training & testing*

We implemented a function for CNN training to optimize the performance for each category. The process begins with model instantiation and compilation (given the high-capacity architecture).

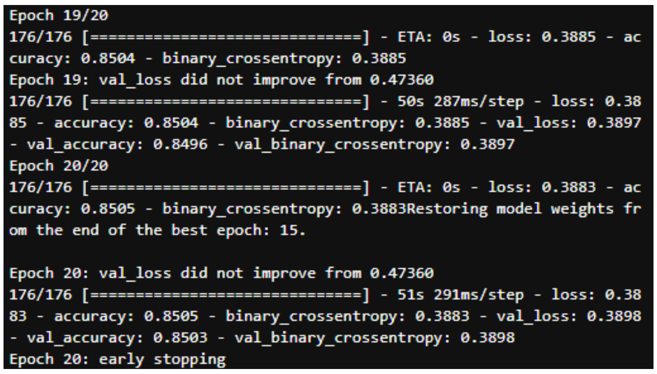
- **Optimizer**: ADAM
  - Why? Because we are dealing with binary outputs that indicate presence/absence of a voxel in the 3D shape
- **Loss function**: binary cross-entropy
  - Why? Because we are dealing with binary outputs that indicate presence/absence of a voxel in the 3D shape
- **Monitoring Metrics**: accuracy and binary cross-entropy
  - Why? For binary voxel outputs we need to measure not only validation accuracy but binary cross entropy that deals with presence/absence of data.

Training is guided by callbacks, including **Early Stopping** to stop training when validation performance does not change and **Model Checkpoints** to save the best-performing model during training. These mechanisms ensure efficiency and prevent overfitting.

For the Early Stopping and Model Checkpoint we have chose to monitor:

- Validation loss: Because Binary Cross entropy is our primary loss function, so tracking it ensures the model generalizes well on unseen data.
- Why not accuracy? Accuracy is helpful for monitoring but can sometimes plateau or improve slightly even if the model overfits

Model performance is then evaluated by visualizing training and validation accuracy and loss across epochs, providing insights into the model's performance over time. This approach ensures a reliable and consistent model training for all of our categories and datasets.



```
Epoch 19/20
176/176 [==============================] - ETA: 0s - loss: 0.3885 - ac
curacy: 0.8504 - binary_crossentropy: 0.3885
Epoch 19: val_loss did not improve from 0.47360
176/176 [==============================] - 50s 287ms/step - loss: 0.38
85 - accuracy: 0.8504 - binary_crossentropy: 0.3885 - val_loss: 0.3897
- val_accuracy: 0.8496 - val_binary_crossentropy: 0.3897
Epoch 20/20
176/176 [==============================] - ETA: 0s - loss: 0.3883 - ac
curacy: 0.8505 - binary_crossentropy: 0.3883Restoring model weights fr
om the end of the best epoch: 15.

Epoch 20: val_loss did not improve from 0.47360
176/176 [==============================] - 51s 291ms/step - loss: 0.38
83 - accuracy: 0.8505 - binary_crossentropy: 0.3883 - val_loss: 0.3898
- val_accuracy: 0.8503 - val_binary_crossentropy: 0.3898
Epoch 20: early stopping
```

Fig 14. *CNN model training for the category 'table'*

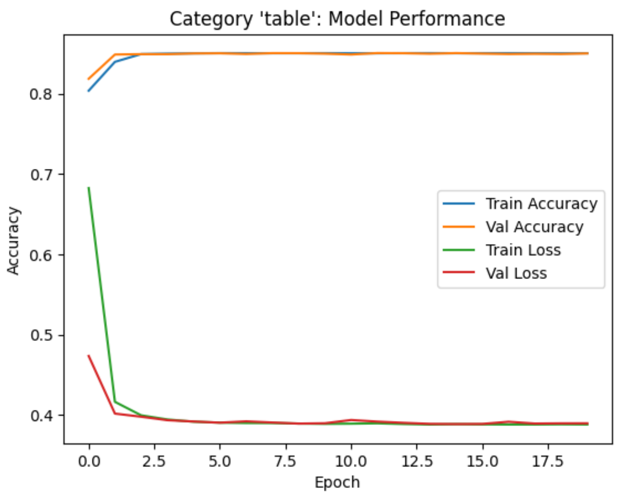
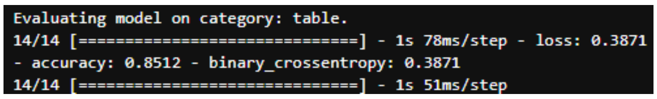


Fig 15. *CNN model performance from the category 'table'*

C. *Models validation & metrics*

Once our models have been trained and we have saved the metrics as binary files for future plotting and comparison. Let's implement and calculate one more metric to measure the performance of our models.

To evaluate the alignment between predicted and original 3D voxels, we implemented the Intersection Over Union (IoU) metric, which quantifies the ratio of the overlap area to the union area between two voxel spaces. IoU serves as an indicator of how accurately the predicted match was.



```
Evaluating model on category: table.
14/14 [==============================] - 1s 78ms/step - loss: 0.3871
- accuracy: 0.8512 - binary_crossentropy: 0.3871
14/14 [==============================] - 1s 51ms/step
```

Fig 16. *CNN evaluation with unseen data on the category 'table'*

Predicted voxels are converted into binary form based on a threshold (0.5), converting all float values from 0 to 1 into True or False. This ensures consistency during comparison. The metric is calculated for individual voxel pairs and averaged across batches to provide a comprehensive performance measure.

For further analysis, IoU values are visualized as a distribution, highlighting instance-level variability and the overall average for the validation set as a horizontal line. Let's remember that this validation subset comprises 5% of the data untouched during training (unseen data for the models).

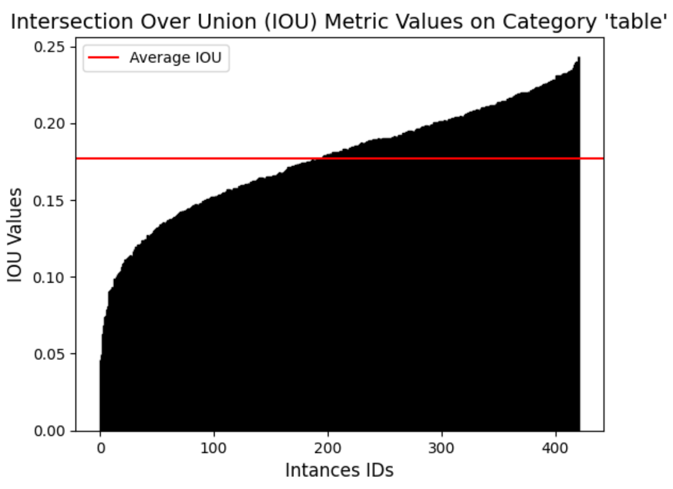


Fig 17. IOU metric for each instance of the 'table' model evaluation set

As a final metric, we implemented a method for visual comparisons by exporting both the predicted and original voxels into OBJ files. Then we loaded images from these OBJ files, allowing us to visualize the results and make visual comparisons based on human perception to assess similarities and differences. This provided qualitative insights into the model's performance by enabling direct visual inspection.

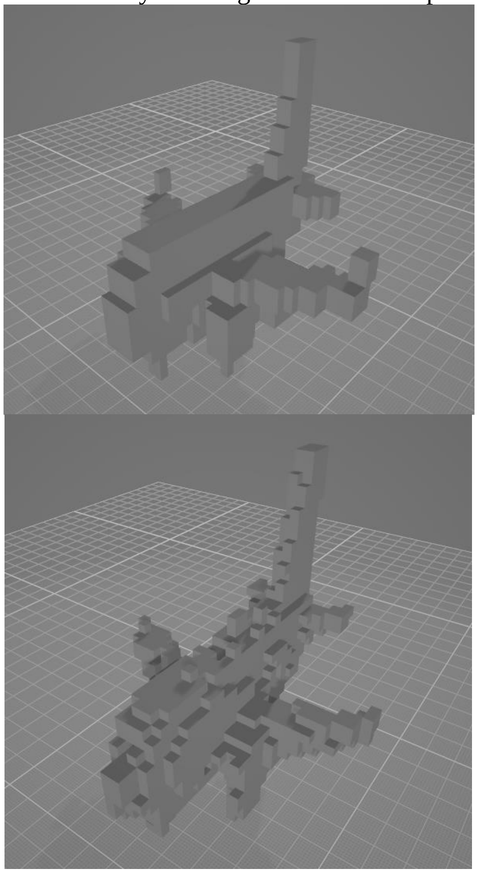


Fig 18. Visual comparison for sample instance from unseen data (top view: original; bottom: predicted) – Sample 1

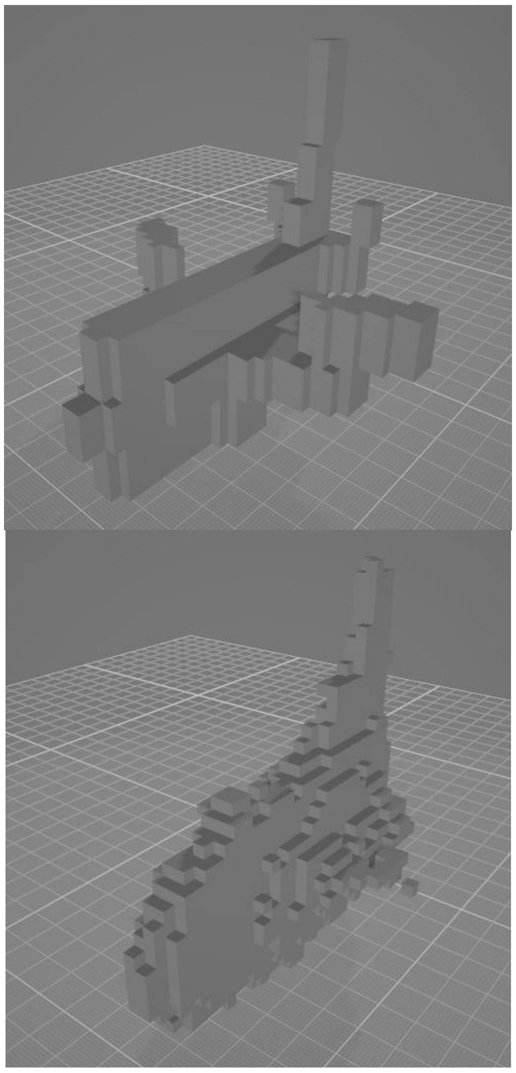


Fig 19. Visual comparison for sample instance from unseen data (top view: original; bottom: predicted) – Sample 2

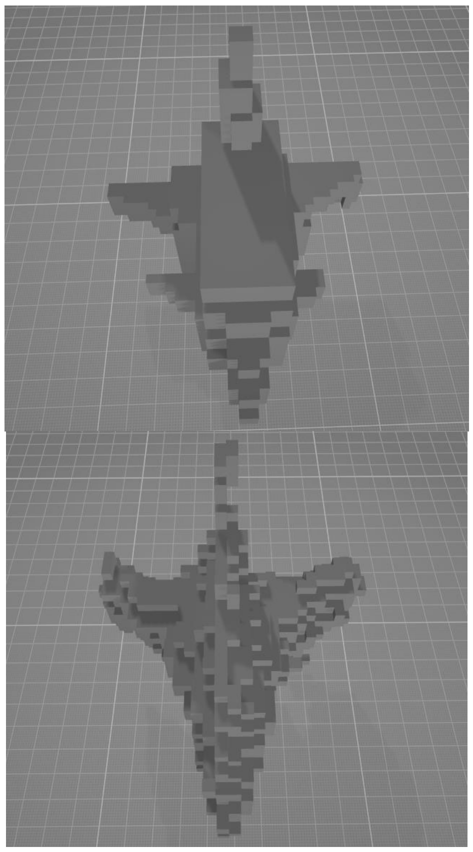


Fig 20. Visual comparison for sample instance from unseen data (top view: original; bottom: predicted) – Sample 3

D. Results from all high-capacity models

Let’s explore the results that we got from all high-capacity models corresponding to the classes with most amounts of data presented previously.

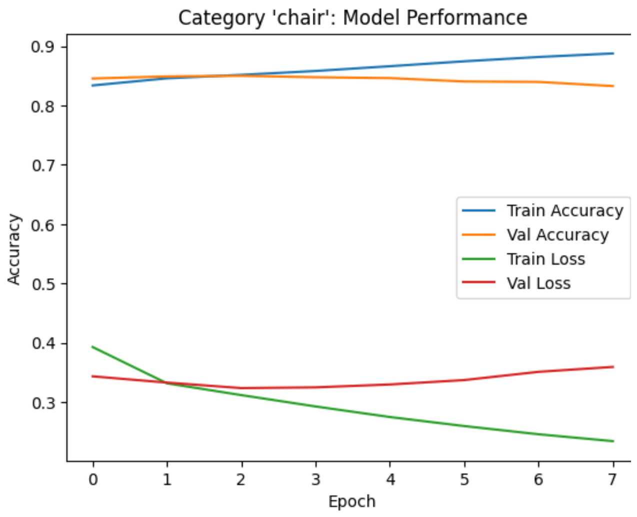


Fig 21. CNN model performance from the category ‘chair’

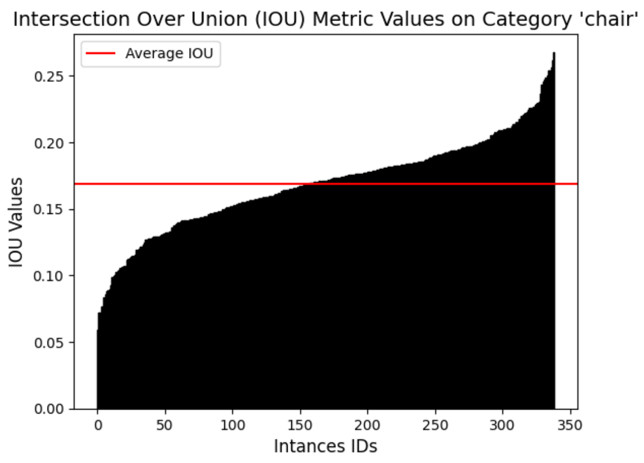


Fig 22. IOU metric results for the category ‘chair’

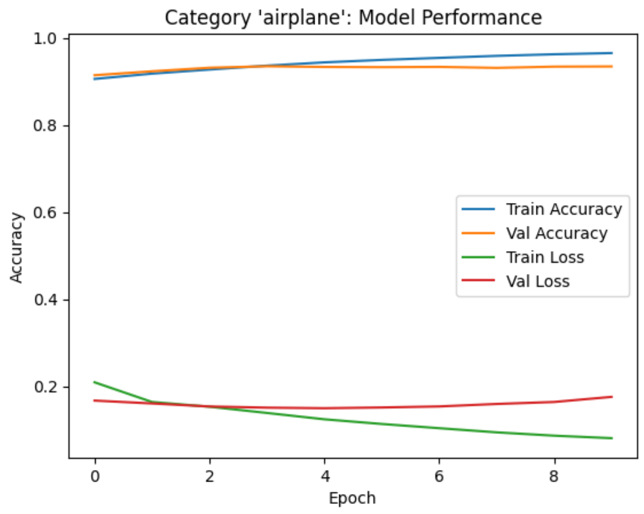


Fig 23. CNN model performance from the category ‘airplane’

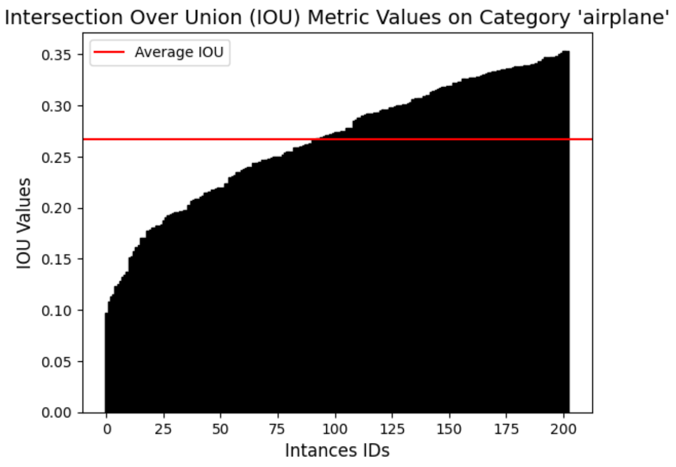


Fig 24. IOU metric results for the category ‘airplane’

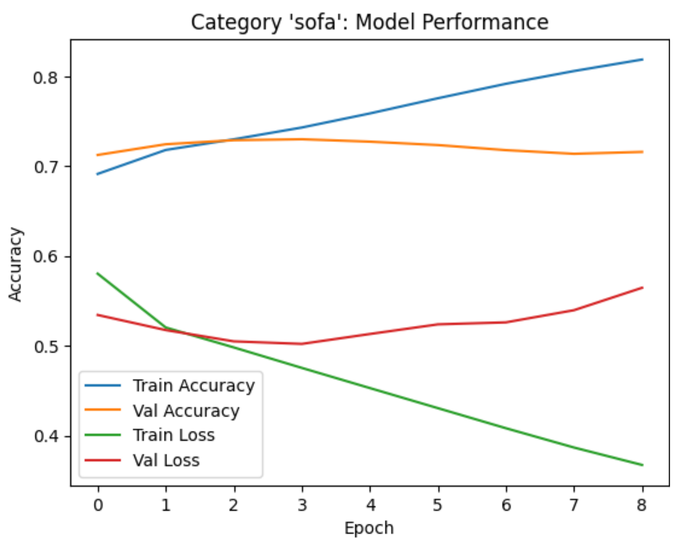


Fig 25. CNN model performance from the category 'sofa

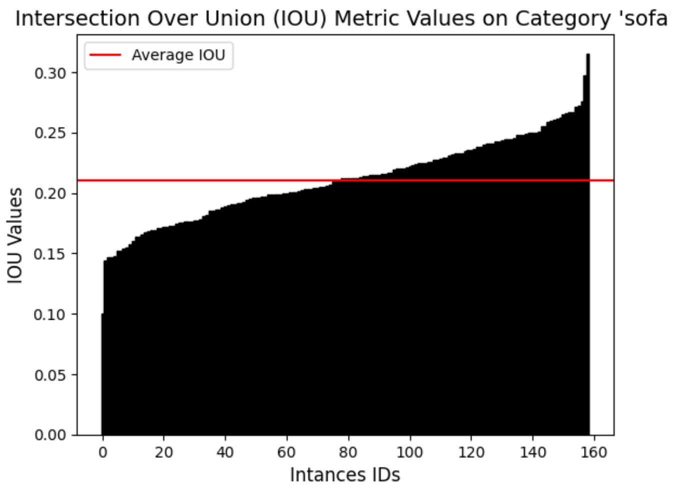


Fig 26. IOU metric results for the category 'sofa'

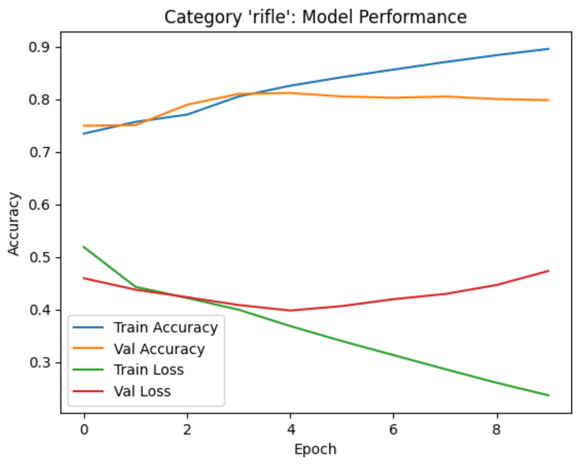


Fig 27. CNN model performance from the category 'rifle'

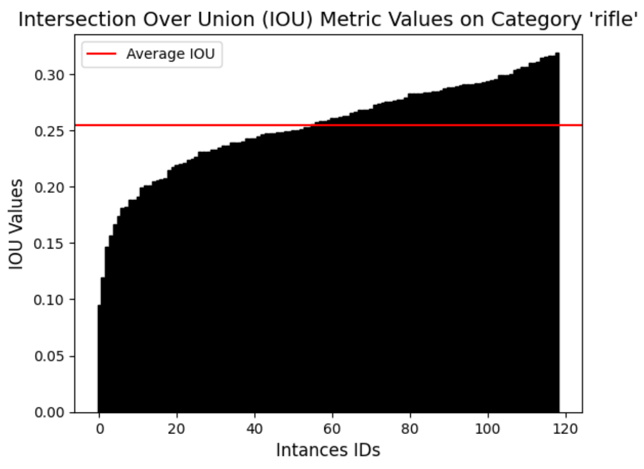


Fig 28. IOU metric results for the category 'rifle'

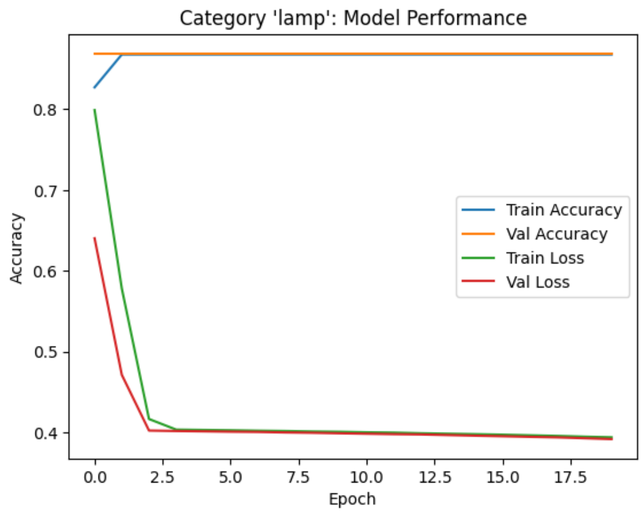


Fig 29. CNN model performance from the category 'lamp'

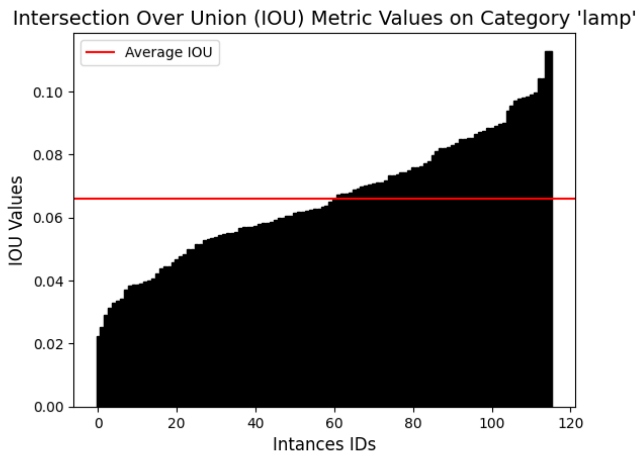


Fig 30. IOU metric results for the category 'lamp'

Finally, to compare our results across models and different categories of 3D objects, we plotted a multi-bar plot to visualize the different metrics calculated across models.

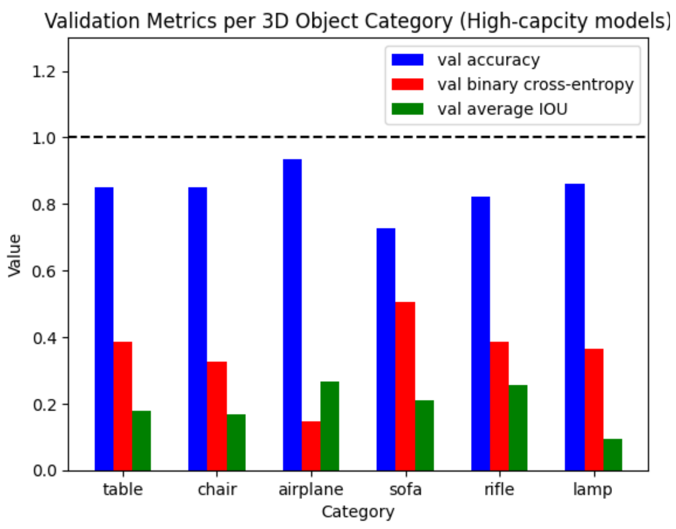


Fig 31. Comparing high-capacity models performance across all metrics

From these results we gained the following **insights**:

- The ‘**airplane**’ category demonstrates the highest accuracy but exhibits the lowest binary cross-entropy and average IoU, suggesting potential overfitting.
- In contrast, the ‘**sofa**’ category presents the most stable metrics, with minimal variation between accuracy, loss, and IoU, indicating a well-generalized model.
- The ‘**lamp**’ category shows a concerning pattern of high accuracy coupled with very low IoU, suggesting overfitting issues.
- Despite these discrepancies, **all categories** display notable performance overall, showcasing the effectiveness of the approach.

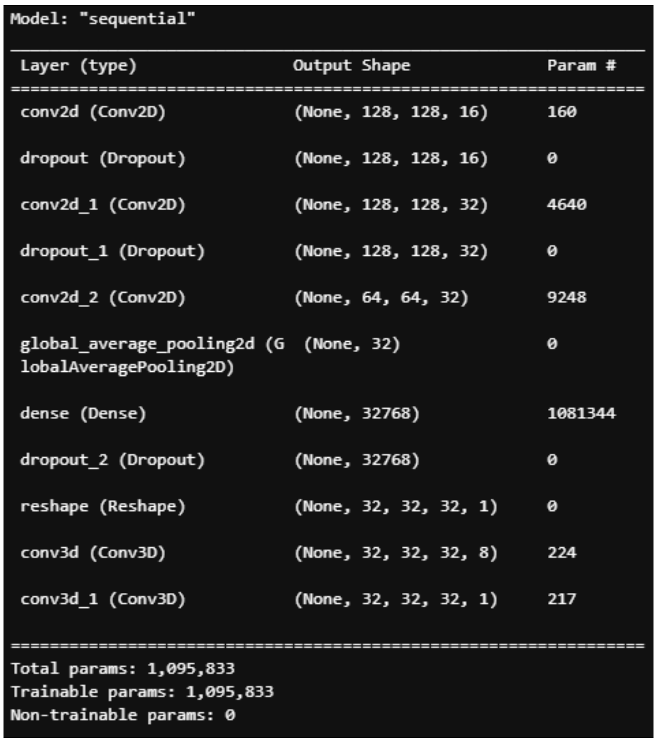
V. Low-Capacity CNNs: Developing the CNN Architecture, Training and Testing

For categories with less data (less than 2,000 instances), we designed a low-capacity CNN architecture to prevent overfitting. This architecture incorporates two complexity reduction techniques:

- **Node Dropouts**: we applied 20% node dropout twice in the 2D convolutional layers and 30% dropout once in the 3D convolutional layers reducing model parameters and enhancing generalization
- **L2 Regularization**: This was used to encourage smaller weights, promoting simplicity and improving model generalization.

This carefully engineered architecture enabled efficient training on smaller datasets. The following figure shows our final low-capacity model architecture which contains:

- **1,095,833 trainable parameters**



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 128, 128, 16)      160

 dropout (Dropout)           (None, 128, 128, 16)      0

 conv2d_1 (Conv2D)           (None, 128, 128, 32)      4640

 dropout_1 (Dropout)         (None, 128, 128, 32)      0

 conv2d_2 (Conv2D)           (None, 64, 64, 32)        9248

 global_average_pooling2d (G  (None, 32)               0
 lobalAveragePooling2D)

 dense (Dense)               (None, 32768)             1081344

 dropout_2 (Dropout)         (None, 32768)             0

 reshape (Reshape)           (None, 32, 32, 32, 1)     0

 conv3d (Conv3D)             (None, 32, 32, 32, 8)     224

 conv3d_1 (Conv3D)           (None, 32, 32, 32, 1)     217

=================================================================
Total params: 1,095,833
Trainable params: 1,095,833
Non-trainable params: 0
```

Fig 32. Low-capacity CNN architecture layers

Given this new CNN architecture, we proceeded with the same data splitting strategy, model training, testing, and validation calculating the defined metrics. These are the extensive results of the low-capacity models.

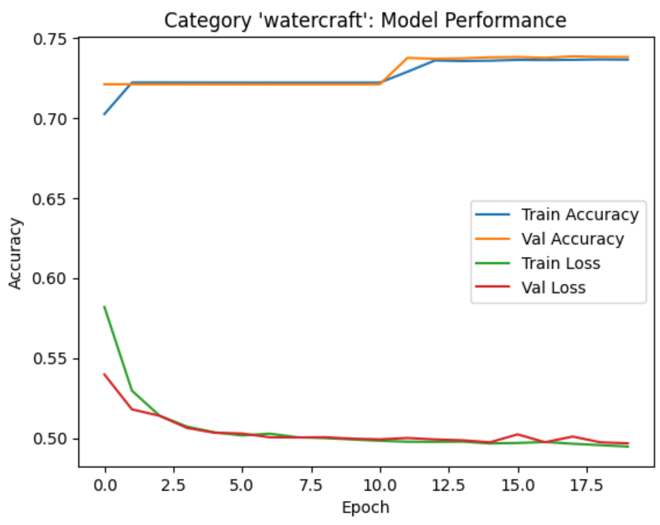


Fig 33. CNN model performance from the category ‘watercraft’

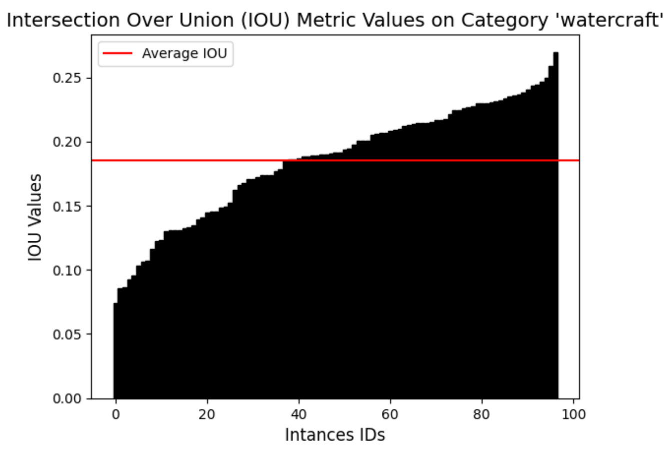


Fig 34. IOU metric results for the category 'watercraft'

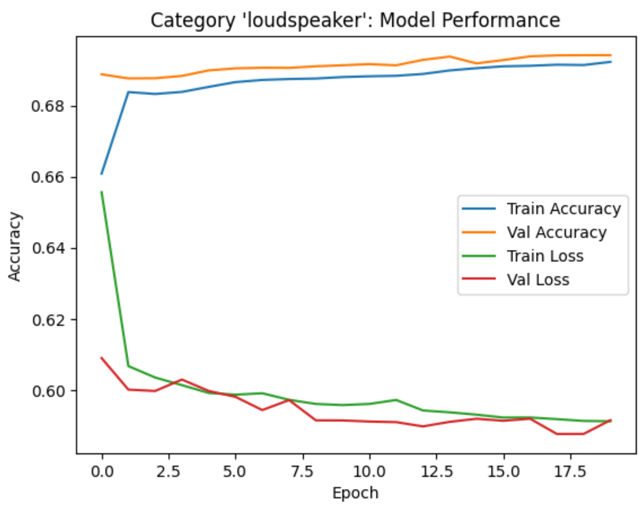


Fig 37. CNN model performance from the category 'loudspeaker'

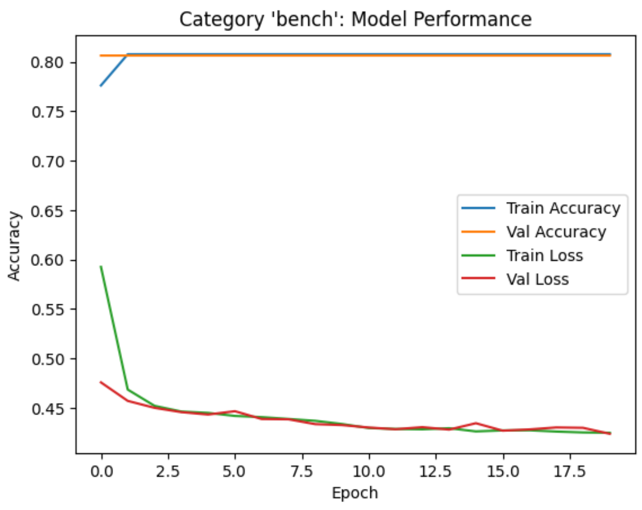


Fig 35. CNN model performance from the category 'bench'

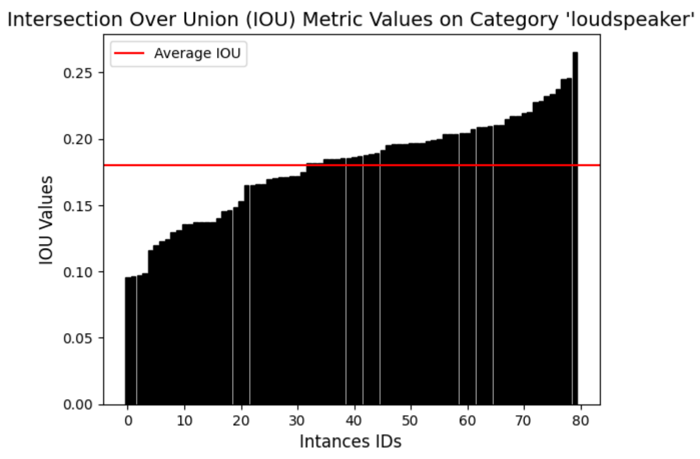


Fig 38. IOU metric results for the category 'loudspeaker'

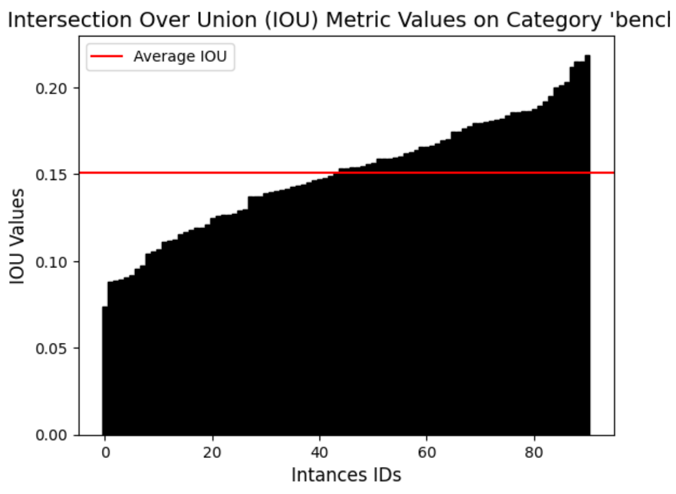


Fig 36. IOU metric results for the category 'bench'

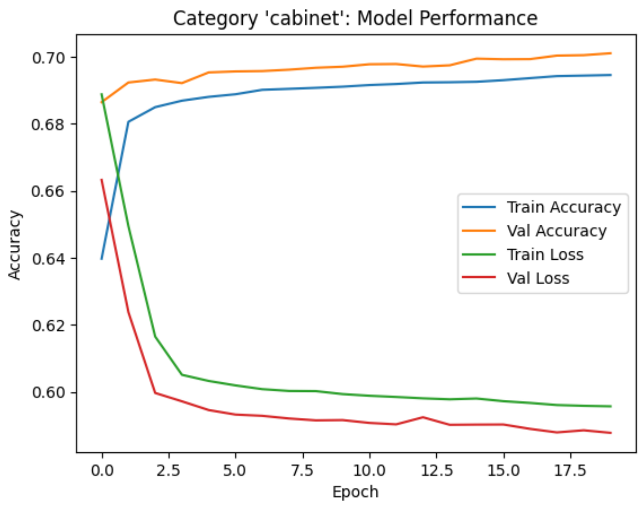


Fig 39. CNN model performance from the category 'cabinet'

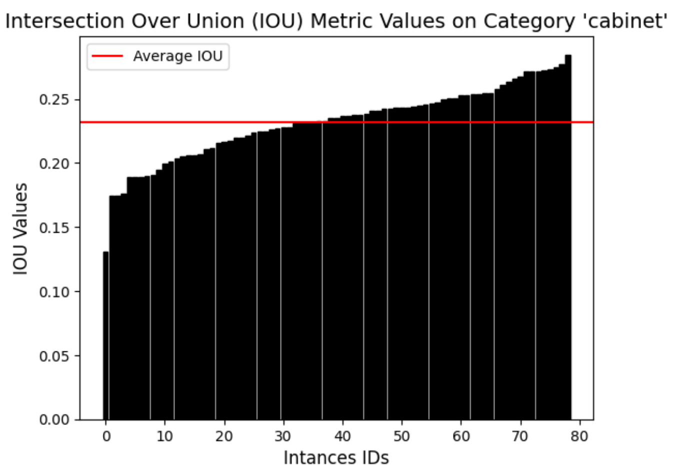


Fig 40. IOU metric results for the category 'cabinet'

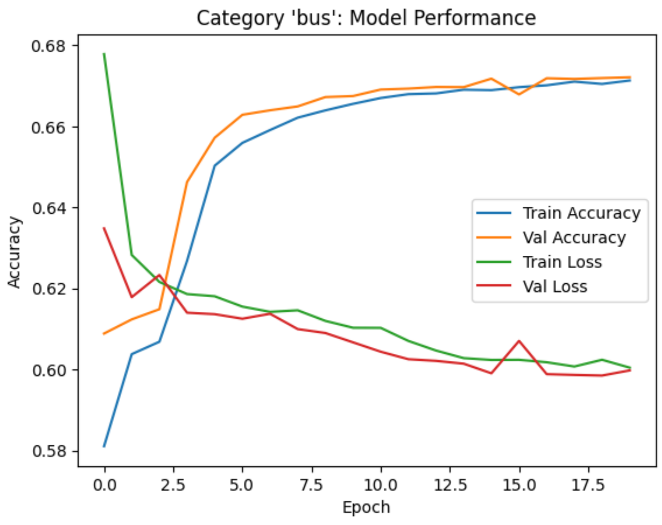


Fig 43. CNN model performance from the category 'bus'

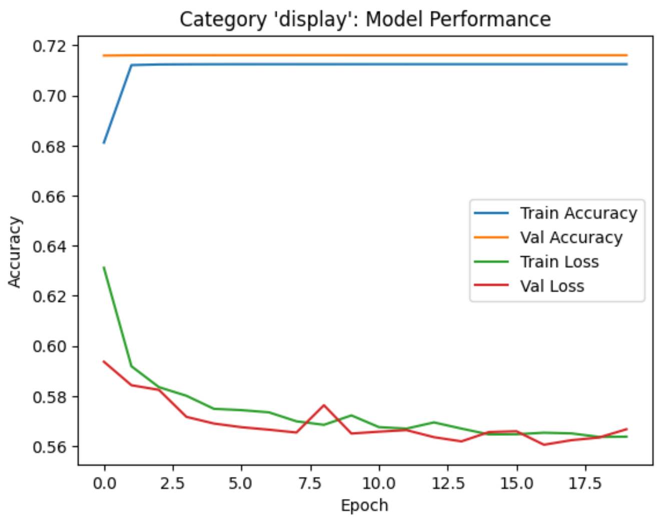


Fig 41. CNN model performance from the category 'display'

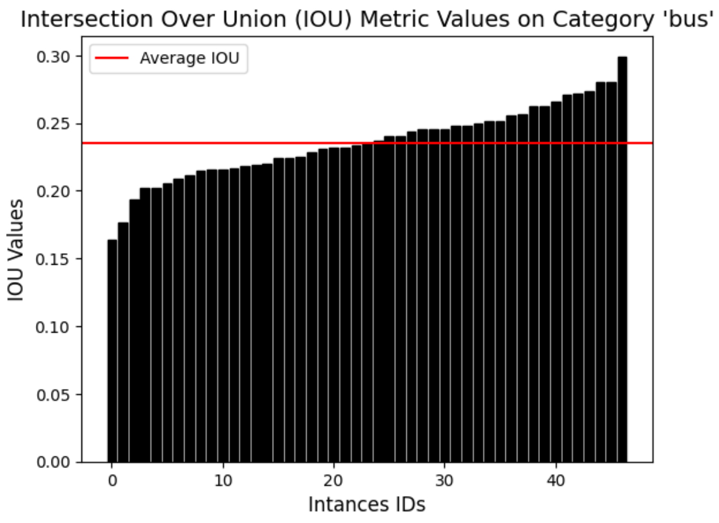


Fig 44. IOU metric results for the category 'bus'

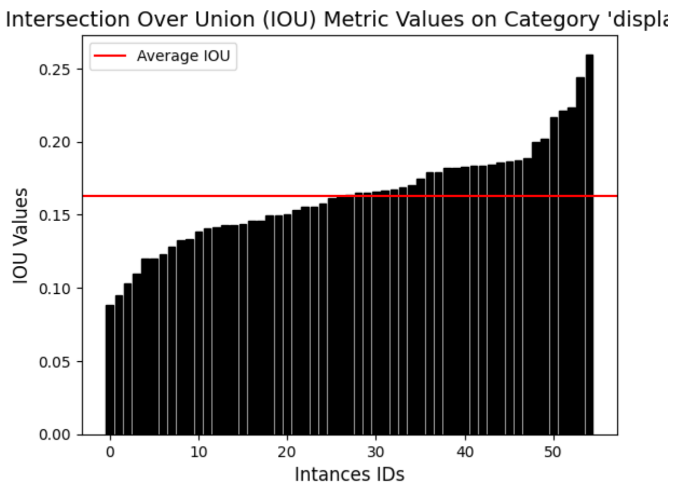


Fig 42. IOU metric results for the category 'display'

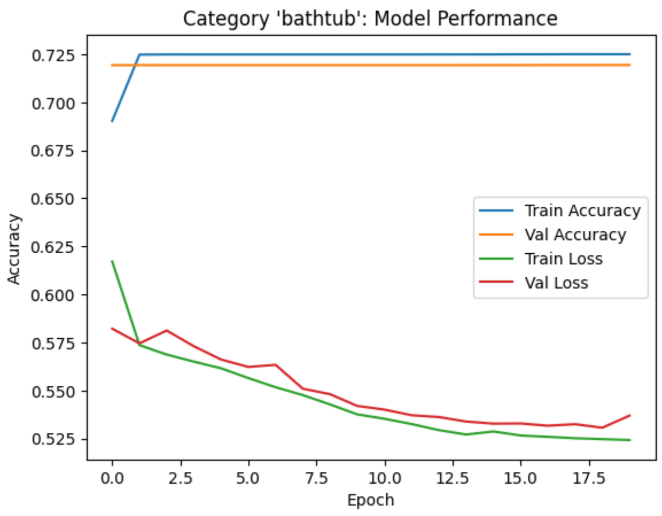


Fig 45. CNN model performance from the category 'bathtub'

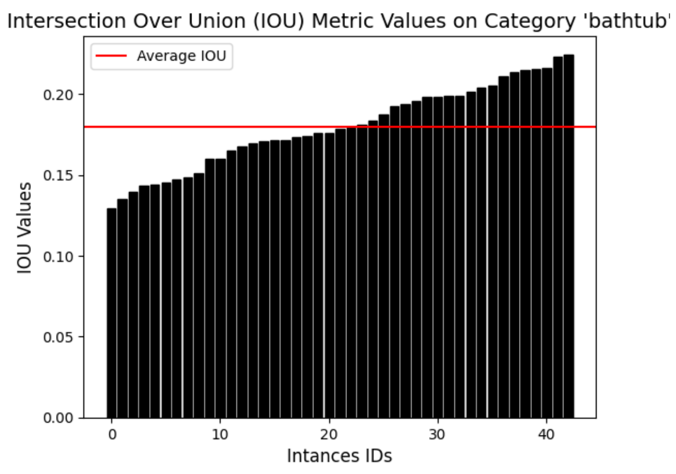


Fig 46. IOU metric results for the category 'bathtub'

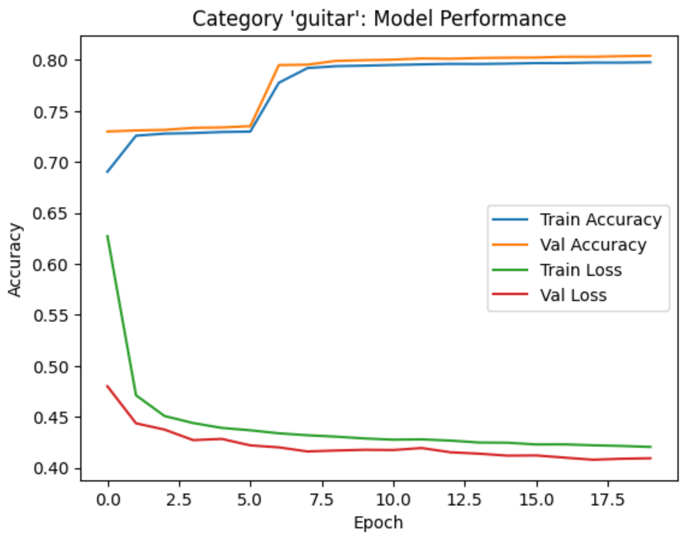


Fig 49. CNN model performance from the category 'guitar'

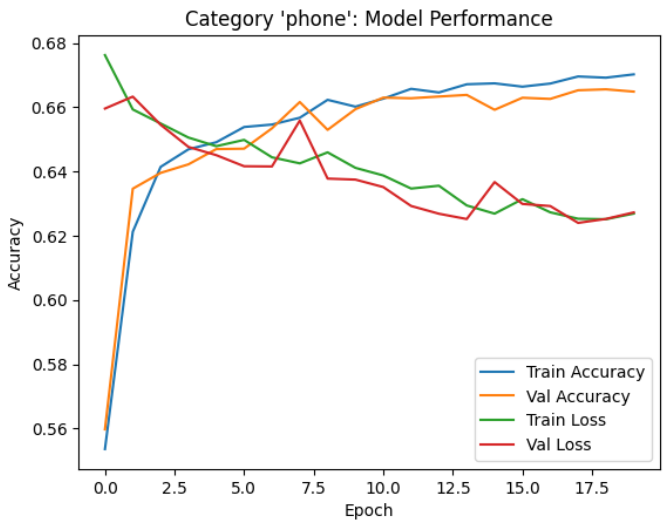


Fig 47. CNN model performance from the category 'phone'

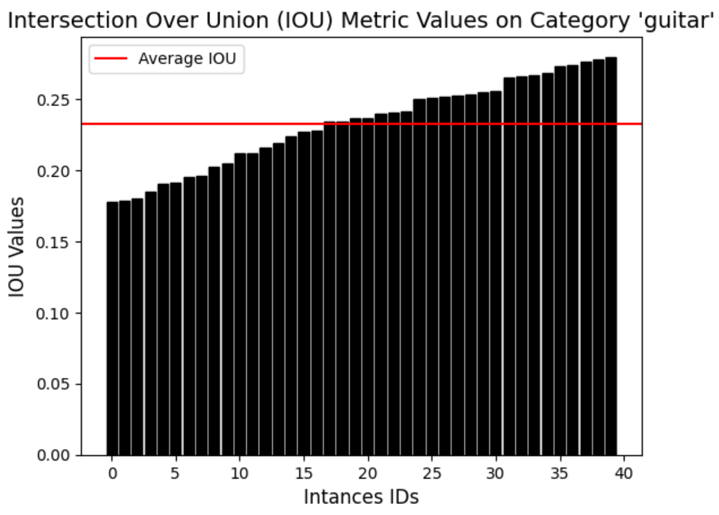


Fig 50. IOU metric results for the category 'guitar'

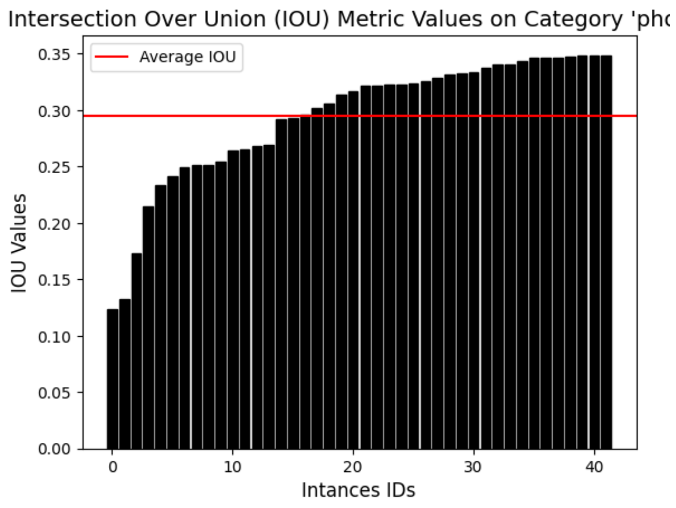


Fig 48. IOU metric results for the category 'phone'

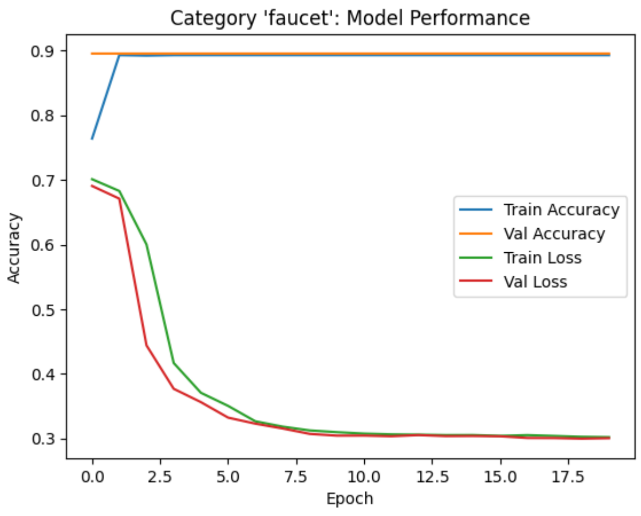


Fig 51. CNN model performance from the category 'faucet'

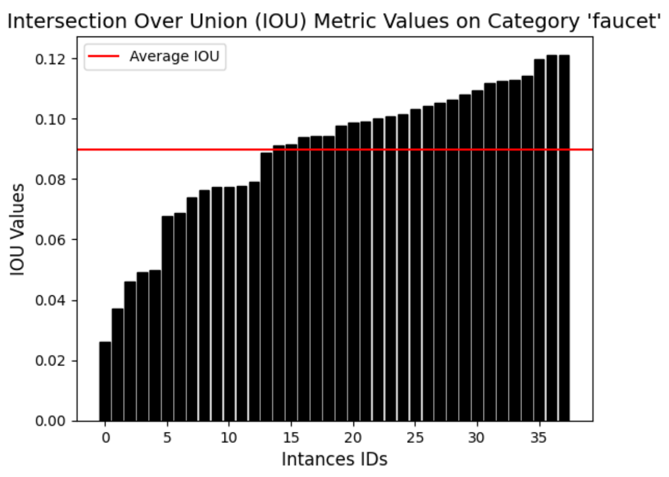


Fig 52. IOU metric results for the category 'faucet'

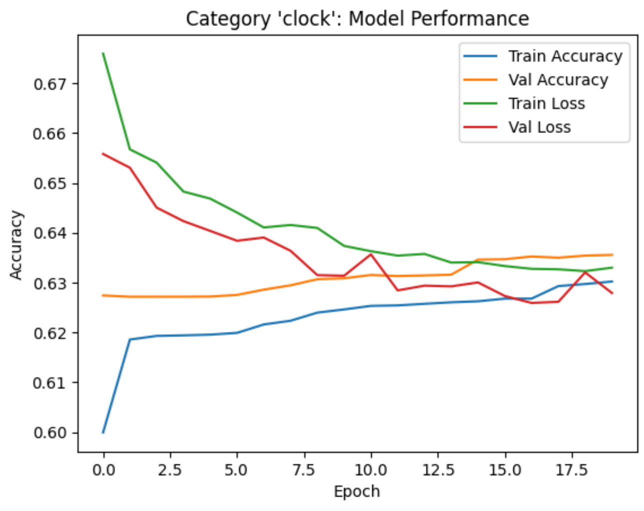


Fig 53. CNN model performance from the category 'clock'

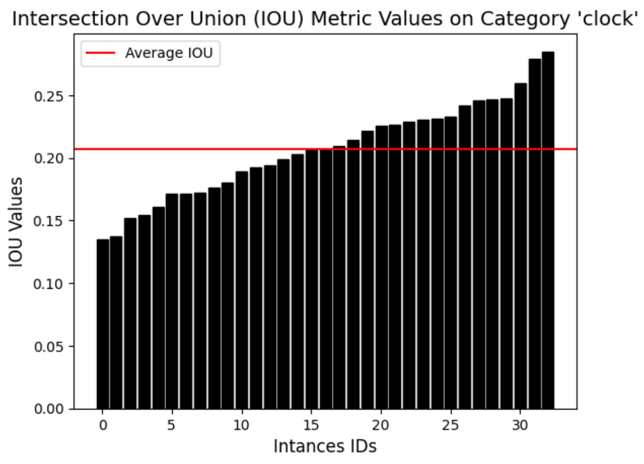


Fig 54. IOU metric results for the category 'clock'

The evaluation of low-capacity models revealed important insights about their performance and our CNN layers approach.

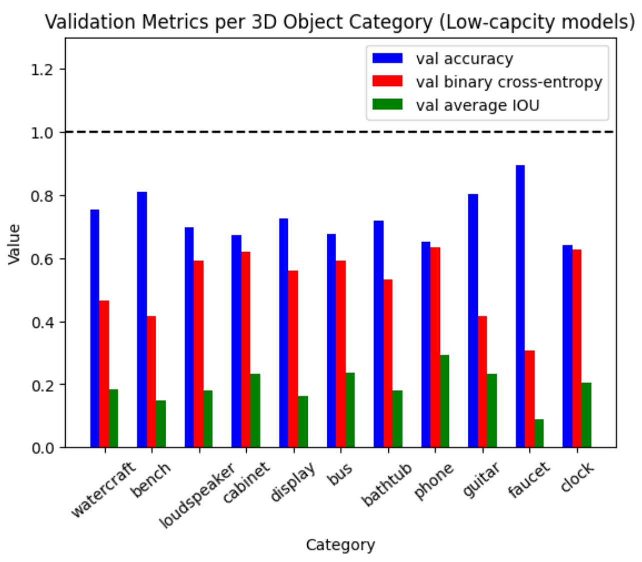


Fig 55. Comparing low-capacity models performance across all metrics

From the plot we learnt the following **insights**:

- There was a notable **increase in validation loss** (binary cross-entropy) across all models.
- We notice a **collective reduction in validation accuracy** compared to high-capacity models, this was expected due to the use of dropout and regularization techniques to reduce the models' complexity.
- However, the average Intersection Over Union (**IoU**) show **slight improvement**, highlighting the models' ability to generalize.
- **5 models** achieved an **IoU** greater than **0.2**, indicating promising overlap between predictions and original 3D objects.

VI. Web Application Development

We decided to develop a web application to make our model predictions easily accessible to the users. The website allows users to upload an image in PNG or JPG format, process it through any of our high-capacity or low-capacity models, and visualize the 3D prediction in an interactive interface. Users can also download the result as an OBJ file for further use.

A. *Technology Stack*

- **Front-End Development:** We used Streamlit 1.14 *[7]* to construct a responsive, and user-friendly interface that simplifies user interaction and allows for a simplistic front-end development.

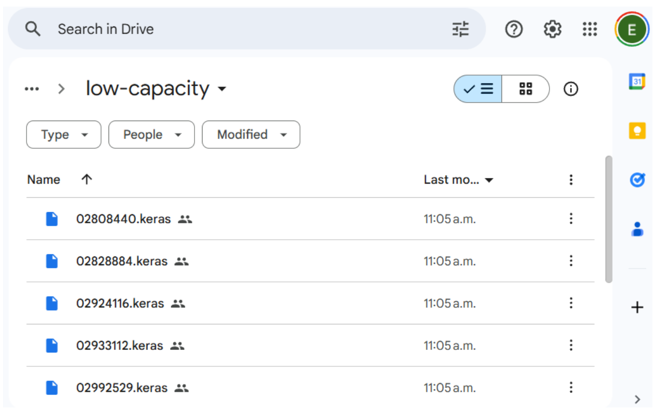
Fig 56. Streamlit logo [7]

- **3D Visualization:** Incorporated the PyVista *[9]* library for smooth and accurate rendering of 3D predictions. This allows users to visualize the results immediately after processing the prediction and use them dynamically for 3D visualization.

Fig 57. PyVista logo [9]

- **Model File Hosting:** Utilized Google Drive for efficient hosting of model files, enabling quick access for download with the **Gdown** Python library. These models can be automatically downloaded by running our script download_models.py in Python to download models.

Fig 58a. Google Drive logo


Search in Drive

low-capacity

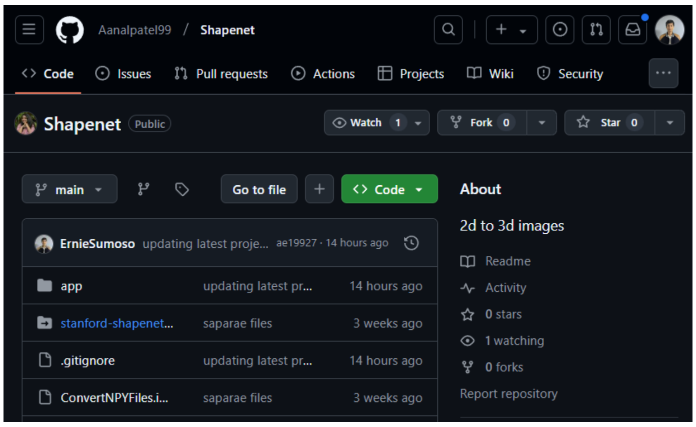
Type | People | Modified

Name	Last mo...
02808440.keras	11:05 a.m.
02828884.keras	11:05 a.m.
02924116.keras	11:05 a.m.
02933112.keras	11:05 a.m.
02992529.keras	11:05 a.m.

Fig 58b. Uploaded models into our Google Drive One

- **Source Code and Version Control:** We also employed GitHub *[8]* for source code and version control. This tool allowed us for collaboration between team members, and easy updates. Our project can be currently downloaded and successfully run by cloning **our GitHub repository** *[12]* and following the Readme instructions.

Fig 59a. GitHub logo [8]



ErnieSumoso updating latest proje... ae19927 · 14 hours ago

app	updating latest pr...	14 hours ago
stanford-shapenet...	saparae files	3 weeks ago
.gitignore	updating latest pr...	14 hours ago
ConvertNPYFiles.i...	saparae files	3 weeks ago

About

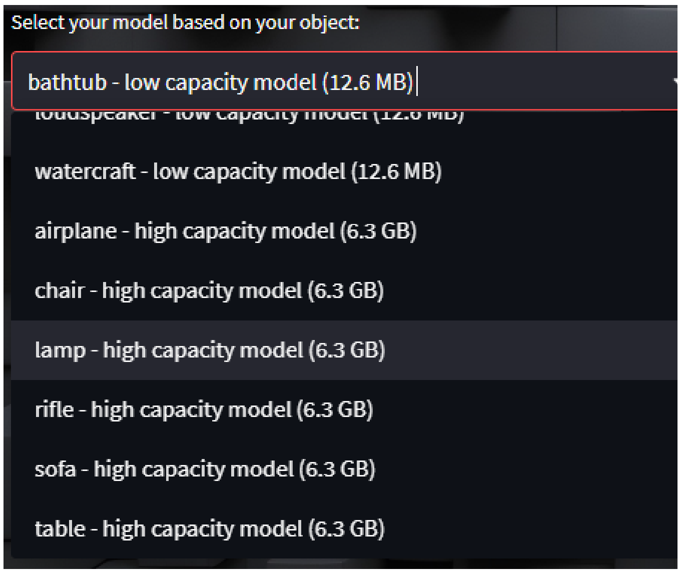
2d to 3d images

Readme
Activity
0 stars
1 watching
0 forks
Report repository

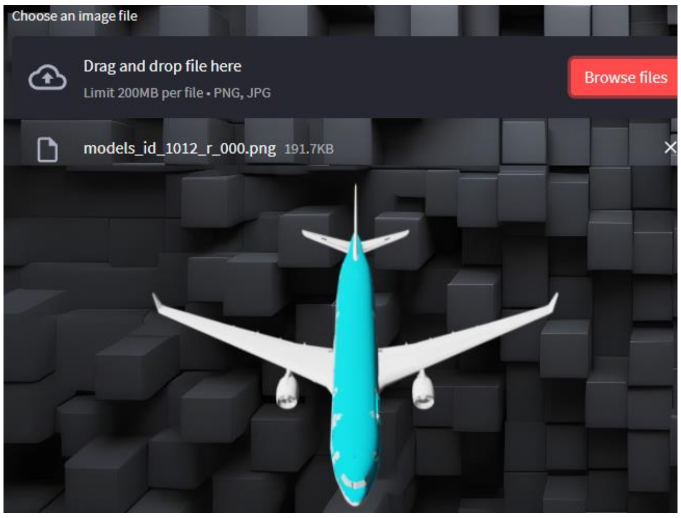
Fig 59b. Our project GitHub repository [12]

B. *Software Features*

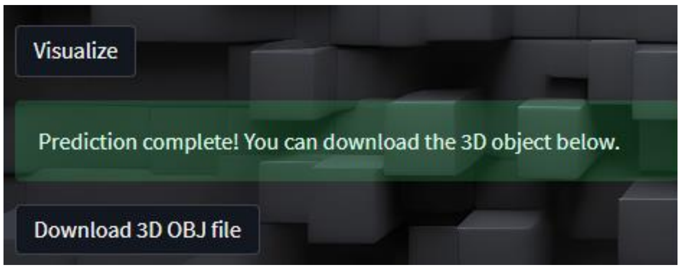
1. **Model Selection:** Users can choose between high-capacity models for detailed and precise predictions from certain object types. Or they can choose between low-capacity models for other object types but using lower resource demands. This flexibility ensures the application meets diverse requirements, from high-precision to low-precision tasks to low-precision tasks.


Select your model based on your object:
bathtub - low capacity model (12.6 MB)
loudspeaker - low capacity model (12.6 MB)
watercraft - low capacity model (12.6 MB)
airplane - high capacity model (6.3 GB)
chair - high capacity model (6.3 GB)
lamp - high capacity model (6.3 GB)
rifle - high capacity model (6.3 GB)
sofa - high capacity model (6.3 GB)
table - high capacity model (6.3 GB)
Fig 60. Model selection feature

2. **Image Upload:** The platform supports PNG and JPG file uploads, with an intuitive section for drag-and-drop or manual file selection. Real-time validation ensures that the file meets specifications, preventing processing errors.


Choose an image file
Drag and drop file here
Limit 200MB per file • PNG, JPG
Browse files
models_id_1012_r_000.png 191.7KB
Fig 61. Image-upload - feature

3. **3D Prediction:** The application processes the uploaded image to generate a 3D voxel-based representation using the selected model. If enabled, our back-end will utilize GPU resources to accelerate the processing time.



Prediction complete! You can download the 3D object below.

Fig 62. Voxel Grid prediction - feature

4. **Visualizing the Result:** Results are displayed, allowing users to explore the 3D model with intuitive mouse controls for rotating, zooming, and panning. Real-time rendering ensures a smooth visualization experience.

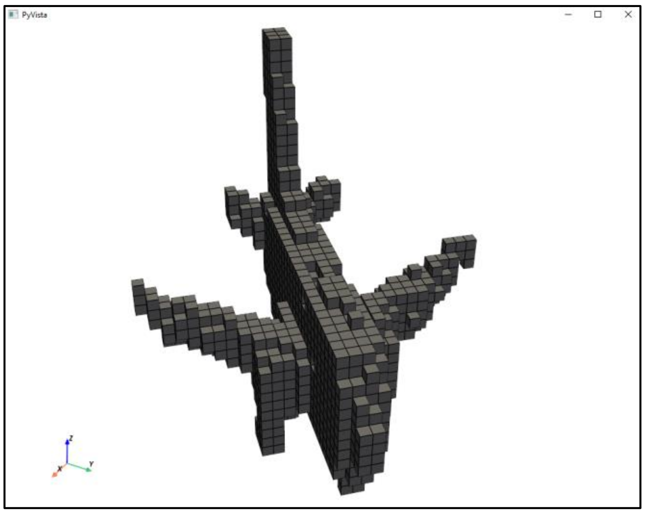


Fig 63. Voxel Grid visualization using PyVista - feature

5. **Downloading the Result:** Finally, users can download the 3D model as an OBJ formatted file. This extension is widely supported and compatible with popular 3D software.

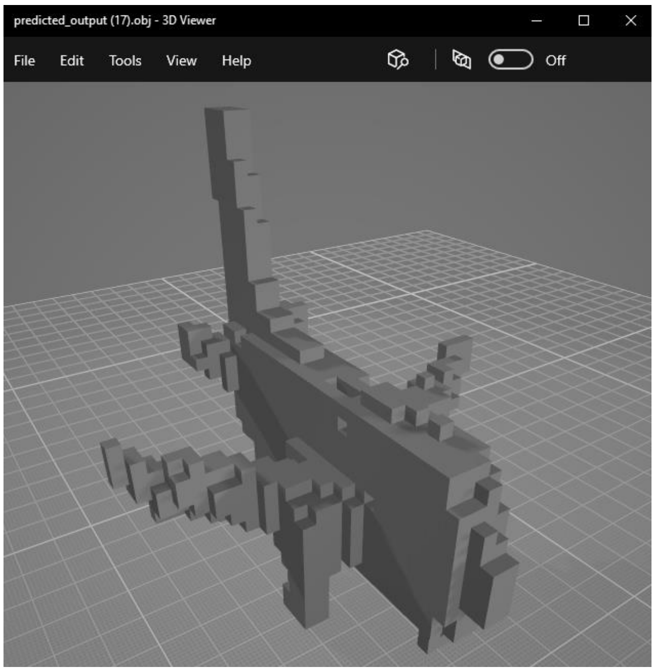
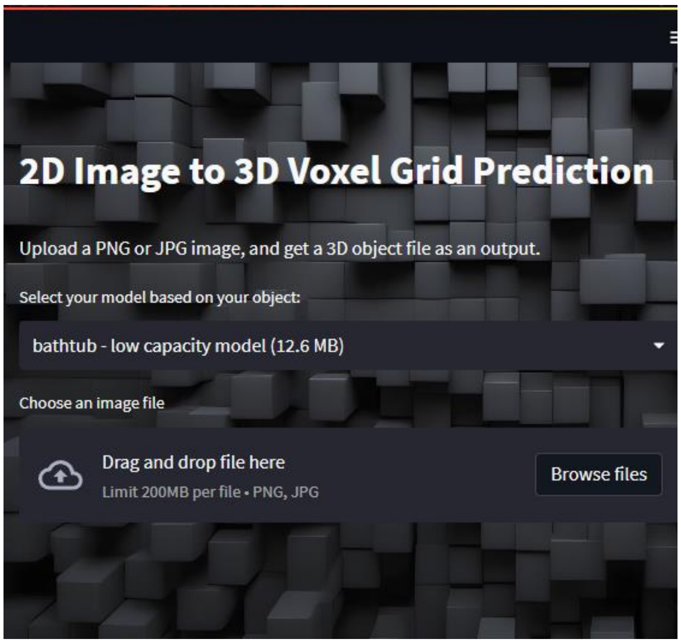


Fig 64. Voxel Grid download as OBJ file- feature

After completing the web development, we performed multiple manual tests, identifying and resolving bugs through a trial-and-error approach. Our final product is presented in the following figure.


2D Image to 3D Voxel Grid Prediction

Upload a PNG or JPG image, and get a 3D object file as an output.

Select your model based on your object:

bathtub - low capacity model (12.6 MB)

Choose an image file

Fig 65. Website main UI – final product

VII. Problems and Challenges Faced

A. Hardware Limitations

The major challenge was our limited hardware resources, particularly RAM and GPU capacity, which affected the processing pipeline of high-capacity models. Let's remember that our data overall consisted in 500,000 images, 50'000 three-dimensional objects. Even though, during training we loaded only part of the data (each class data), still our resources were not enough to keep them on RAM and train a CNN with over 500 million parameters (high-capacity models).

- **Solution 1**: We **distributed the computing tasks** amongst our team members resources. During the image generation pipeline, we all contributed to the generation of these images by running the pipelines on a segment of the entire data.

- **Solution 2:** For training purposes, we **bought extra hardware** capacity as **32 GB of RAM**. This was enough to keep the 6GB CNN on RAM (as well as the data) during training.

- **Solution 3:** We bought the Google Drive One subscription to amplify our storage space to 100 GB. This allowed us to upload our model files (approximately 40 GB) and download them anywhere.

B. Time-complexity Limitations

Processing large datasets and 3D predictions proved to be time-intensive, especially with high-capacity models and large amounts of data. Some of our data pipelines took consecutive 20+ hours to finish. Meanwhile, when we first trained a CNN, it was taking so long that we stopped the process and came up with a different approach for training high-capacity CNNs.

- **Solution**: We integrated **Nvidia CUDA** *[10]* **and cuDNN** *[11]* to fully **leverage GPU resources**, significantly accelerating CNN training. This integration took as an entire day to figure out and set everything up. However, it proved extremely useful as the **training times improved 10 times faster** than the original CPU training. This allowed us to parallelize computations and handle larger workloads efficiently, **reducing processing times from days to manageable hours** duration.

C. Dependencies version Limitations and Windows support

Managing compatibility between different library versions proved to be a challenge during the latest stages of the project. Particularly with libraries like TensorFlow *[5]* and Streamlit *[7]*, we were not able to develop a front-end without breaking our models loading code. We resolved these issues by carefully aligning dependency versions and testing the environment.

- **Solution**: We opted to **downgrade TensorFlow to version 2.10.0** *[5]*, as it was the last version to offer full Windows support while enabling integration with CUDA *[10]* and cuDNN *[11]* for GPU acceleration. Then, we also **adjusted Streamlit to version 1.14.0** *[7]*, aligning it with the chosen TensorFlow version for compatibility. Fortunately, we did not have further problems with other libraries.

VIII. Conclusion

In this work, we approached the problem of predicting 3D object voxel grids from 2D images by developing and testing high-capacity and low-capacity convolutional neural networks (CNN). The dataset was divided into three subsets based on the number of instances, allowing us to apply different models that matched the characteristics of each category's data. For categories with a large number of instances, we used **high-capacity CNNs** (570 million parameters). On the other hand, for categories with fewer instances, we employed **low-capacity models** (1 million parameters) designed with specific techniques to prevent overfitting, such as **node dropouts and L2 regularization**.

The results from our experiments show that **high-capacity models** performed well on large datasets. These exhibit high accuracy but occasionally **suffer from overfitting**, as evidenced by **lower IoU scores**. For example, the **'airplane'** category achieved the best accuracy but had the lowest average IoU, suggesting that the model was overfitted. In contrast, categories such as **'sofa'** show more stable metrics with less variation, indicating a well-balanced model able to generalize with unseen data.

For categories with less than 2000 instances, we engineered a low capacity CNN architecture applying the following techniques for regularization: **node dropouts** (20% twice for 2D convolutional layers and 30% once for 3D convolutional layers) and **L2 regularization**, encouraging better generalization. The results show a decrease in validation accuracy due to the reduced capacity. However, they also show a **slight improvement in average IoU scores**, indicating increase capability of generalizing predictions on unseen data.

Finally, we implemented a **front-end** with multiple features to make our models and predictions easily accessible to potential users, displaying our work and final product.

In summary, this study demonstrated the **importance of designing CNN architectures based on the data characteristics and complexity**. We highlighted the trade-offs between model capacity, accuracy, metrics and generalization. Our results also show the **value of custom evaluation metrics like IoU** for assessing model performance in 3D object prediction tasks. Our findings suggest that a **balance between model complexity and regularization techniques** are essential for achieving accurate predictions and avoiding overfitting when facing large amounts of imbalanced data.

IX. Future Work

For future work we recommend further improving the performance of the 3D object voxel prediction models by exploring:

- **Different voxel resolutions:** We only tested with a resolution of (32, 32, 32)
- **Different image resolutions:** We only tested with a resolution of (128, 128)
- **Data augmentation on 3D data:** This is a way of solving class imbalance. Particularly by applying 3D rotations to the voxel grids, which will help generate more diverse training samples and address class imbalance by artificially increasing the size of underrepresented classes.

These further experiments will provide valuable insights into improving generalization and model performance, enhancing our 3D object recognition approach.

References

[1] Kasten, Y., Doktofsky, D., & Kovler, I. (2020). End-To-End Convolutional Neural Network for 3D Reconstruction of Knee Bones from Bi-planar X-Ray Images. In Lecture notes in computer science (pp. 123–133). https://doi.org/10.1007/978-3-030-61598-7_12

[2] Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., & Yu, F. (2015, December 9). ShapeNet: an Information-Rich 3D model repository. arXiv.org. https://arxiv.org/abs/1512.03012

[3] Panmari. (n.d.). GitHub - panmari/stanford-shapenet-renderer: Scripts for batch rendering models using Blender. Tested with models from stanfords shapenet library. GitHub. https://github.com/panmari/stanford-shapenet-renderer

[4] Blender Foundation. (n.d.). blender.org - Home of the Blender project - Free and Open 3D Creation Software. blender.org. https://www.blender.org/

[5] TensorFlow Developers. (n.d.). TensorFlow: An end-to-end open-source machine learning platform. TensorFlow. Retrieved December 9, 2024, from https://www.tensorflow.org

[6] PyTorch. (n.d.). PyTorch. https://pytorch.org/

[7] Streamlit. (n.d.). Streamlit: The fastest way to build and share data apps. Streamlit. Retrieved December 9, 2024, from https://streamlit.io

[8] GitHub. (n.d.). GitHub: Where the world builds software. GitHub. Retrieved December 9, 2024, from https://github.com

[9] PyVista Developers. (n.d.). PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK). PyVista. Retrieved December 9, 2024, from https://pyvista.org

[10] NVIDIA. (n.d.-a). CUDA Toolkit. NVIDIA. Retrieved December 9, 2024, from https://developer.nvidia.com/cuda-toolkit

[11] NVIDIA. (n.d.-b). cuDNN: NVIDIA CUDA Deep Neural Network library. NVIDIA. Retrieved December 9, 2024, from https://developer.nvidia.com/cudnn

[12] Patel, A., Saravanan, J., Diaz, D., & Sumoso, E. (2024, Decemeber). Shapenet. GitHub. Retrieved December 9, 2024, from https://github.com/Aanalpatel99/Shapenet