Université Paris Saclay

3D Data Analysis

Master of Research in Artificial Intelligence and Machine Vision

Lab

3D Shape Retrieval using View-Based Descriptors

Student:	*Supervisor:*	*Master:*
Yann Terrom	Hedi Tabia	M2MMVAI

Academic Year
2023-2024

Contents

List of Figures

Listings

1 Introduction

1.1 Objective of the Project

The primary objective of this project is to develop an algorithm for 3D shape retrieval using View-Based Descriptors. But before delving into the specifics, let's first understand what 3D shape retrieval entails.

1.1.1 Understanding 3D Shape Retrieval

3D shape retrieval is the process of searching and retrieving three-dimensional models from a database based on their shape similarity. In other words, given a new 3D model, the goal is to find the most similar shape within a reference database. This task plays a crucial role in various applications, enabling efficient and effective content-based retrieval in 3D model databases.

1.1.2 Applications of 3D Shape Retrieval

The applications of 3D shape retrieval are diverse and impactful. Some notable examples include:

- **Computer-Aided Design (CAD):** Engineers and designers can benefit from quickly finding similar 3D models for inspiration or modification.
- **Biomedical Imaging:** Identifying and matching anatomical structures in medical imaging can aid in diagnosis and treatment planning.
- **Virtual and Augmented Reality:** Enhancing virtual environments by retrieving 3D objects that closely resemble real-world counterparts.
- **Robotics:** Matching 3D shapes can facilitate object recognition and manipulation for robots in diverse environments.

1.1.3 Project Focus

In this project, our focus is on utilizing View-Based Descriptors for 3D shape retrieval. The aim is to develop a system that, given a new 3D shape, can efficiently find the most similar shapes from a pre-existing database. We will employ the trimesh library for handling 3D mesh data and PyTorch for building and training a neural network model to achieve this objective.

1.2 Dataset Overview

For this project, we have chosen to utilize the McGill dataset, available at `https://www.cim.mcgill.ca/~shape/benchMark/`. The dataset comprises various classes of 3D models, each belonging to distinct categories, such as ants, crabs, hands, humans, octopuses, pliers, snakes, spectacles, spiders, and teddy bears.

The dataset structure is organized as follows:

- **Root Level:** The main directory of the dataset.
- **Second Level:** Ten subdirectories, each representing a different class of 3D models (e.g., ants, crabs, hands, humans, octopuses, pliers, snakes, spectacles, spiders, teddy bears).
- **Third Level:** For each class, two subdirectories are present - one containing files with the "Im" extension and the other with the "Ply" extension. The naming convention is consistent; for example, for the "humans" class, we have "humansIm" and "humanPly" directories.
- **Fourth Level:** Within the "Ply" directories, we find the 3D model files in the PLY format. These files are compressed in a gzip format (.gz).

1.2.1 Dataset Structure Diagram

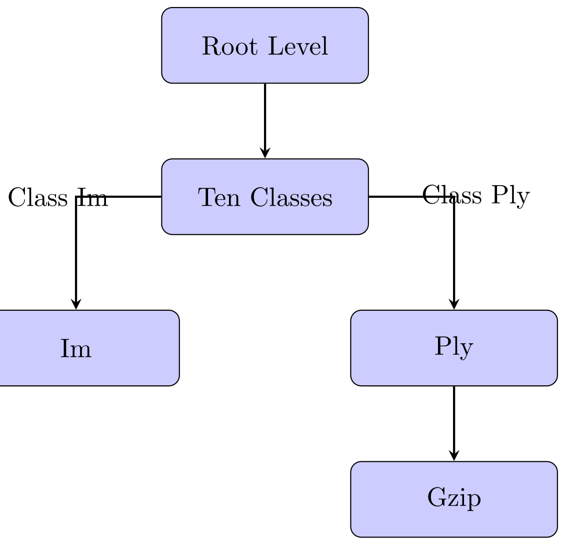
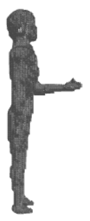


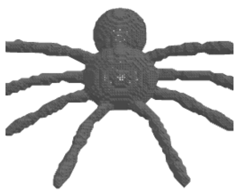
Figure 1: Structure of the McGill Dataset

1.2.2 Example Images

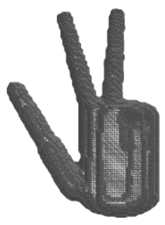
Let's illustrate a few examples by displaying 3D models from the dataset:



(a) Human



(b) Spider



(c) Hand

(d) Octopus

Figure 2: Example 3D Models from Different Classes

In Figure 2, we showcase example 3D models from various classes in a 2x2 grid format.

1.3 Plan of Action

To address the complexity of directly comparing 3D models, our approach involves extracting view-based descriptors. Instead of comparing the 3D models directly, we will generate 2D views (images) from the 3D models at different camera angles. Subsequently, these 2D views will be passed through a convolutional neural network (CNN) to extract features, specifically a latent vector.

1.3.1 Generating Latent Vectors

The process involves the following steps:

1. **3D to 2D Views:** From each 3D model, generate multiple 2D views by varying the camera angles.
2. **CNN Feature Extraction:** Pass the 2D views through a CNN to obtain a latent vector for each image. This vector captures high-level features.
3. **Vector Latent Space:** The latent vectors represent the 3D model in a feature-rich latent space.

1.3.2 Dataset Modification

To implement this approach, we will modify the structure of our dataset. For each class (e.g., "humans"), we will add two new components:

- **classnameImage2D:** A directory to store the 2D images generated from the 3D models.
- **latent_vector.json:** A JSON file storing the latent vectors associated with each 3D model.

The latent vectors, representing the features extracted from the 2D views, will be paired with their corresponding 3D models in the dataset.

1.3.3 Shape Retrieval Process

When presented with a new 3D model for shape retrieval, the process involves:

1. **Generate 2D Views:** Create 2D views from the new 3D model at various angles.
2. **Extract Latent Vectors:** Pass the generated 2D views through the pre-trained CNN to obtain the latent vectors.
3. **Search in Latent Space:** Compare the latent vectors with those in the reference dataset, identifying the closest match based on distance metrics.

Here, the latent vectors serve as our descriptors for comparing and retrieving 3D shapes.

In conclusion, with the outlined methodology of 3D shape retrieval using view-based descriptors, the modified dataset structure, and the proposed shape retrieval process, we are equipped with a comprehensive plan to commence our work. The combination of 2D views, CNN-based feature extraction, and latent vectors provides a promising foundation for efficiently comparing and retrieving 3D shapes from our reference dataset.

1.4 Configuration Environment

In this project, the choice of the computing environment plays a crucial role in achieving efficient and effective results. While platforms like Google Colab offer substantial computational power, their limitations, such as the lack of support for rendering trimesh and OpenGL, hinder certain aspects of the project.

For this reason, a dedicated local setup has been employed. The development environment is an Acer Predator Triton 300, running Ubuntu 22.04. The system is equipped with a GeForce RTX 3060 graphics card, providing ample computational capabilities. This configuration ensures smooth execution of programs, particularly those involving neural networks, and facilitates the rendering of 3D models for further analysis.

2 Generating Reference Database

To build our reference database for 3D shape retrieval, the first step is to create 2D views from the 3D models and save them. This not only facilitates faster experimentation by executing the program only once but also efficiently manages memory usage. We will utilize a custom dataset class, 'McGillDataset3D', to load and organize our McGill dataset.

2.1 McGillDataset3D Class

The 'McGillDataset3D' class is designed to handle the loading and organization of the McGill dataset. Here is an overview of its functionality:

```python
class McGillDataset3D(Dataset):
    def __init__(self, root_dir: str, file_type: str = 'ply', seed=42):
        """
        Custom dataset for McGill dataset.

        Parameters:
        - root_dir (str): Root directory of the dataset.
        - file_type (str, optional): Type of file to load data from. Defaults to '
    ply'.
        - seed (int, optional): Seed for shuffling. Defaults to None.
        """
        self.root_dir = root_dir
        self.file_type = file_type

        # List all the categories (subdirectories) in the root directory
        self.categories = os.listdir(root_dir)

        # Create a dictionary to map category names to their respective
    subdirectories
        self.category_paths = {category: os.path.join(root_dir, category) for
    category in self.categories}

        # Get paths to all the data files in the dataset
        self.all_data_paths = self.get_all_data_paths()

        # Shuffle the data paths for randomness
        random.seed(seed)
        random.shuffle(self.all_data_paths)
```

Listing 1: McGillDataset3D Class

This class provides a structured way to access and manage the McGill dataset, listing categories, mapping category names to paths, and shuffling data paths for later use.

2.2 Generating 2D Views

The generate 2d views method in the McGillDataset3D class is responsible for creating 2D views from a given 3D shape by rotating it around both the X and Y axes. Here is a detailed breakdown of its functionality:

```python
def generate_2d_views(self, idx, num_steps_x=5, num_steps_y=5, resolution=(256,
    256)):
    """
    Generate 2D views from a 3D shape by rotating around both X and Y axes.

    Parameters:
    - idx (int): Index of the 3D shape in the dataset.
    - num_steps_x (int, optional): Number of steps in rotation around the X axis.
    Defaults to 5.
    - num_steps_y (int, optional): Number of steps in rotation around the Y axis.
    Defaults to 5.
    - resolution (tuple, optional): Resolution of the generated images. Defaults to
     (256, 256).

    Returns:
```

```python
    - views (list): List of PIL Image objects representing 2D views.
    """
    # Load the data (3D mesh) using the existing __getitem__ method
    sample = self.__getitem__(idx)

    mesh = sample['data']

    # Create a scene with the mesh
    scene = mesh.scene()

    # Initialize a list to store generated views
    views = []

    # Calculate step sizes for rotation
    step_size_x = 360 / num_steps_x
    step_size_y = 180 / num_steps_y

    # Iterate through rotation angles
    for angle_x in range(0, 360, int(step_size_x)):
        for angle_y in range(-90, 90, int(step_size_y)):
            # Rotation matrix around the X-axis
            rotate_x = trimesh.transformations.rotation_matrix(
                angle=np.radians(angle_x), direction=[1, 0, 0], point=scene.centroid)

            # Rotation matrix around the Y-axis
            rotate_y = trimesh.transformations.rotation_matrix(
                angle=np.radians(angle_y), direction=[0, 1, 0], point=scene.centroid)

            # Combine the rotations
            rotate_combined = trimesh.transformations.concatenate_matrices(rotate_x, rotate_y)

            # Apply the combined transform to the camera view transform
            camera_old, _geometry = scene.graph[scene.camera.name]
            camera_new = np.dot(rotate_combined, camera_old)
            scene.graph[scene.camera.name] = camera_new

            # Render the scene and save the image
            try:
                # Save a render of the object as a PNG

                png = scene.save_image(resolution=resolution, visible=False)

                # Convert the PNG to a PIL Image
                image = Image.open(io.BytesIO(png))

                # Append the image to the views list
                views.append(image)
            except BaseException as e:
                print(f"Unable to save image: {str(e)}")

    return views, sample
```

Listing 2: generate 2d views Method

This method takes an index (idx) corresponding to a 3D shape in the dataset and generates a series of 2D views by rotating the shape around both the X and Y axes. The number of steps for rotation (num steps x and num steps y) and the resolution of the generated images (resolution) can be customized. The resulting list views contains PIL Image objects representing the generated 2D views.

2.3 Processing the Dataset and Generating 2D Views

The ‘$process_dataset$‘ function is responsible for iterating through the entire dataset, generating 2D views for each 3D shape, and saving them in their corresponding directories. Here is an in-depth explanation of its functionality:

```python
def process_dataset(dataset, num_steps_x=6, num_steps_y=6):
    """
    Process a dataset, generate 2D views, and save them to the corresponding
    directories.

    Parameters:
    - dataset: The dataset to process.
    - num_steps_x: The number of steps along the x-axis for generating 2D views.
    - num_steps_y: The number of steps along the y-axis for generating 2D views.
    """

    for i in tqdm(range(len(dataset)), desc="Processing dataset"):
        sample = dataset[i]

        # Extract the relevant part of the path
        path = sample['path']
        category = sample['label']
        base_path = os.path.join(dataset.root_dir, category)

        # Create the Image2D subdirectory if it doesn't exist
        file_name_without_extension = os.path.splitext(os.path.basename(path))[0]
        file_name_without_extension = os.path.splitext(os.path.basename(
    file_name_without_extension))[0]
        image2d_subdir = os.path.join(base_path, f'{category}Image2D',
    file_name_without_extension)
        os.makedirs(image2d_subdir, exist_ok=True)

        views, sample_data = dataset.generate_2d_views(i, num_steps_x, num_steps_y)

        # Save the images in the image2d_subdir
        for idx, image in enumerate(views):
            image_filename = f"{file_name_without_extension}_view_{idx + 1}.png"
            image_path = os.path.join(image2d_subdir, image_filename)
            image.save(image_path)
        print(f"{image_path} successfully saved ")
```

Listing 3: process_dataset Function

This function iterates through each sample in the dataset, generates 2D views using the '$generate_2d_views$' method, and saves the resulting images in the appropriate subdirectory. For the given configuration with '$num_steps_x = 6$' and '$num_steps_y = 6$', a total of 36 images will be generated for each 3D shape, each with a resolution of (256, 256).

The successful execution of the '$process_dataset$' function demonstrates the effective generation and organization of 2D views from the 3D shapes in the dataset. With these images now available, we are well-prepared to transition to the next phase of our project—building a convolutional neural network (CNN) to extract latent vectors from these 2D views.

3 CNN for Latent Vector Extraction and Classification

In this section, we aim to achieve two objectives simultaneously. The primary goal is to design a Convolutional Neural Network (CNN) capable of effectively extracting latent vectors from 2D views of 3D shapes. Additionally, we leverage the CNN architecture to create a classifier that can predict the class of an object based on its 2D view.

The rationale behind integrating classification into our CNN design is twofold. First, by training the CNN as a classifier, we ensure that the network learns to extract meaningful features (latent vectors) that distinguish between different object classes. Second, the trained classifier serves as a means to streamline the comparison of latent vectors. Instead of comparing a latent vector with all vectors in the reference database, we can selectively compare it only with vectors from the same class.

For instance, when presented with a 2D view generated from a 3D model of a human, the CNN classifier predicts the class (e.g., "human"). We can then use this predicted class label to focus the latent vector comparison on the subset of latent vectors corresponding to humans in our reference database. This targeted approach enhances efficiency in retrieval tasks.

3.1 Training the Classifier

Now, let's delve into the details of training our classifier. We will use a ResNet-18 architecture, a popular pre-trained model known for its effectiveness in image classification tasks. ResNet-18 consists of multiple residual blocks, and its architecture can be summarized as follows:

- Input layer
- Convolutional layers with batch normalization and ReLU activation
- Residual blocks (multiple repetitions)
- Fully connected layer for classification

The classifier code defines a custom PyTorch module named `Classifier`, which loads a pre-trained ResNet-18 model and modifies its fully connected head to match the number of classes in our dataset.

```
class Classifier(nn.Module):
    def __init__(self, num_classes):
        super(Classifier, self).__init__()
        # Load a pre-trained ResNet model
        self.resnet = models.resnet18(pretrained=True)
        # Modify the classifier head to match the number of classes in your dataset
        in_features = self.resnet.fc.in_features
        self.resnet.fc = nn.Linear(in_features, num_classes)

    def forward(self, x):
        return self.resnet(x)
```

Listing 4: Resnet18 classifier

For training, we split our dataset into a training set (for building the reference database) and a test set (for evaluation). The classifier is trained using the Adam optimizer and cross-entropy loss. We iterate through a specified number of epochs, tracking training and test loss, as well as accuracy values.

Here is the training loop:

- Set the device (CPU or GPU) based on availability.
- Initialize the classifier.
- Define the loss function (cross-entropy) and optimizer (Adam).
- Set the number of training epochs.

- Iterate through epochs, separating the process into training and testing phases.
- In the training phase:
  - Set the model to training mode.
  - Iterate through batches in the training loader.
  - Perform a forward pass, compute the loss, and update weights.
  - Record running loss and track predictions and true labels for accuracy computation.
  - Print and store training loss and accuracy for each epoch.
- In the testing phase:
  - Set the model to evaluation mode.
  - Iterate through batches in the test loader.
  - Compute the test loss and track predictions and true labels.
  - Calculate and store test accuracy.
  - Print test loss and accuracy for each epoch.

Here is the training loop:

```python
# Set the device (CPU or GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize the classifier
num_classes = len(custom_dataset.classes)
classifier = Classifier(num_classes).to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(classifier.parameters(), lr=0.001)

# Set the number of training epochs
num_epochs = 10
# Lists to store training and test loss, and training and test accuracy values
train_loss_values = []
train_accuracy_values = []
test_loss_values = []
test_accuracy_values = []

# Training and testing loop
for epoch in range(num_epochs):
    # Training phase
    classifier.train()  # Set the model to training mode
    running_loss = 0.0
    predictions = []
    true_labels = []

    for images, labels in tqdm(train_loader):
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()  # Zero the gradients
        outputs = classifier(images)  # Forward pass
        loss = criterion(outputs, labels)  # Compute the loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights

        running_loss += loss.item()

        _, predicted = torch.max(outputs.data, 1)
        predictions.extend(predicted.cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

    # Calculate training accuracy
    train_accuracy = accuracy_score(true_labels, predictions)

    # Store training loss and accuracy values
    train_loss_values.append(running_loss / len(train_loader))
    train_accuracy_values.append(train_accuracy)

    # Print training loss and accuracy for each epoch
    print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {train_loss_values[-1]},
     Training Accuracy: {train_accuracy * 100:.2f}%")

    # Testing phase
    classifier.eval()  # Set the model to evaluation mode
    test_running_loss = 0.0
    test_predictions = []
    test_true_labels = []

    with torch.no_grad():
        for images, labels in tqdm(test_loader):
            images, labels = images.to(device), labels.to(device)
            outputs = classifier(images)
            test_loss = criterion(outputs, labels)
            test_running_loss += test_loss.item()

            _, predicted = torch.max(outputs.data, 1)
            test_predictions.extend(predicted.cpu().numpy())
            test_true_labels.extend(labels.cpu().numpy())

    # Calculate test accuracy
    test_accuracy = accuracy_score(test_true_labels, test_predictions)

    # Store test loss and accuracy values
    test_loss_values.append(test_running_loss / len(test_loader))
    test_accuracy_values.append(test_accuracy)

    # Print test loss and accuracy for each epoch
    print(f"Epoch {epoch + 1}/{num_epochs}, Test Loss: {test_loss_values[-1]}, Test
     Accuracy: {test_accuracy * 100:.2f}%")
```

Listing 5: Resnet18 classifier training

The training loop concludes by storing the loss and accuracy values, which will be visualized in learning curves for further analysis.

This approach not only trains a classifier for object classification but also prepares the CNN for the dual task of extracting latent vectors from 2D views.

3.2 Result of Classification

Metric	Value (%)
Training Accuracy	99.50
Test Accuracy	93.57

Table 1: Classifier Performance Metrics

After training and evaluating the classifier, the obtained results are promising, demonstrating a high level of performance. The model achieved an impressive accuracy of 99.50% on the training set and maintained strong generalization capabilities with a test accuracy of 93.57%. These metrics indicate the classifier's ability to effectively learn and generalize features from 2D views of our 3D shapes, showcasing its potential for accurately classifying objects in our reference database.

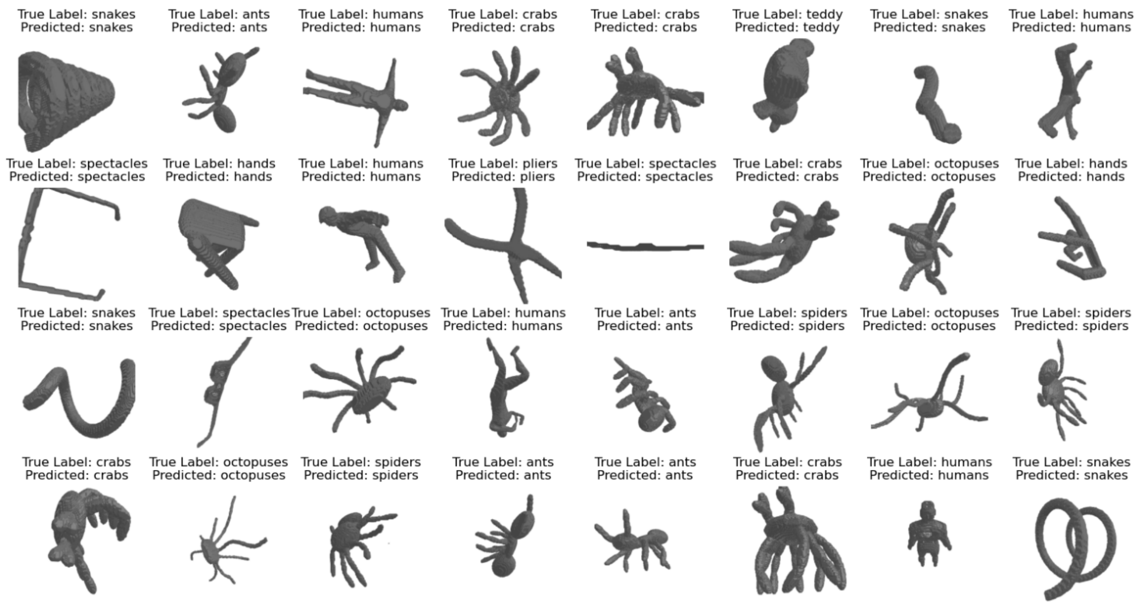
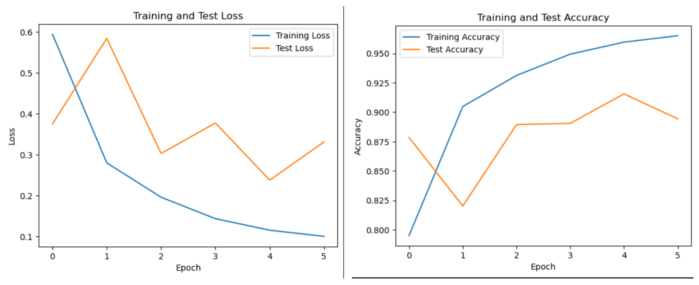


Figure 3: Images true/predicted label Training



(a) Training Loss Curve (b) Accuracy Curve

Figure 4: Learning Curves for Classifier Training

3.3 Feature Extraction for Latent Vector

To perform feature extraction and obtain the latent vector from the 2D views, we utilize the pre-trained ResNet18 model employed in our classifier. The following code snippet illustrates the process:

```python
# Instantiate the classifier model
model = Classifier(num_classes)

# Load the pre-trained model weights
model_path = 'model.pth'
model.load_state_dict(torch.load(model_path, map_location=device))

# Move the model to the appropriate device (GPU or CPU)
model.to(device)

# Create a feature extractor by removing the last fully connected layer
features_extractor = torch.nn.Sequential(*(list(model.resnet.children())[:-1]))
```

Listing 6: Feature Extraction with ResNet18

By removing the final fully connected layer, we retain the feature extraction capabilities of ResNet18, and the output of this modified model now serves as our latent vector for further comparison and retrieval tasks.

3.4 Initialization of Feature Extractor

To prepare for the extraction of latent vectors, we initialize our feature extractor by removing the last layer of the ResNet classifier. This modification ensures that we retain the feature extraction capabilities of the ResNet model while discarding the final classification layer.

```python
features_extractor = torch.nn.Sequential(*(list(model.resnet.children())[:-1]))
```

Listing 7: Feature Extractor Initialization

3.5 Latent Vector Generation

Once the feature extractor is set up, we proceed to generate latent vectors for 3D models in our training dataset. The latent vectors are derived from the 2D views of the models. For each 3D model, we iterate through its associated 2D views, extract the latent vectors using the feature extractor, and store them in a dictionary.

```python
def generate_latent_vectors(train_dataset, features_extractor, transform, device):
    # Iterate through the train_dataset
    for idx in tqdm(range(len(train_dataset))):
        # Access the sample in the dataset
        sample = train_dataset[idx]
        label = sample['label']
        path_2D = sample['path_2d_data_folder']

        # ... (Details of path extraction omitted for brevity)

        # Create or ensure the existence of the 'latent' directory
        latent_directory = os.path.join(to_save_directory, 'latent')
        if not os.path.exists(latent_directory):
            os.makedirs(latent_directory)

        # Dictionary to store latent vectors for each 3D model
        latent_vectors_dict = {}

        # Loop over all images related to one 3D model
        for image_file in image_files:
            # ... (Details of image processing omitted for brevity)

            # Forward pass through the latent vector extractor
            with torch.no_grad():
                latent_vector = features_extractor(input_tensor).cpu().numpy()
            latent_vectors_dict[image_file] = latent_vector.flatten().tolist()

        # Save or append the latent vectors dictionary to the JSON file
        output_json_path = os.path.join(latent_directory, f"{model3d}LatentVector.json")
        with open(output_json_path, 'w') as json_file:
            json.dump(latent_vectors_dict, json_file)
```

Listing 8: Latent Vector Generation

3.6 Analysis

The generation of latent vectors from 2D views provides a compact representation of the 3D models. This process enables efficient retrieval and comparison of shapes in our subsequent retrieval algorithm. The resulting latent vectors, stored in JSON files, form a crucial part of our final 3D shape retrieval database.

4 3D Shape Retrieval Application

With our comprehensive 3D shape database in place, containing 3D models, corresponding 2D views, and their respective latent vectors, we are well-equipped to develop a 3D shape retrieval application. The objective here is to take new 3D models, generate 2D views from them, extract latent vectors, and perform a retrieval process by finding the nearest neighbors in our database.

4.1 Workflow Overview

The retrieval process follows a sequence of steps:

1. **Input**: New 3D model.
2. **Generate 2D Views**: Generate a set of 2D views from the 3D model.
3. **Extract Latent Vectors**: Extract latent vectors from the generated 2D views.
4. **Retrieve Nearest Neighbors**: Calculate distances between the extracted latent vectors and those in the database. Choose the nearest neighbors based on a chosen distance metric.
5. **Match 3D Shapes**: Retrieve and match the 3D shapes associated with the nearest neighbors.

This workflow enables efficient and effective 3D shape retrieval, providing a practical application for various domains.

4.2 Implementation of Distance Metric

To facilitate the retrieval process, we begin by implementing a distance metric, which measures the dissimilarity between two vectors. In our case, we start with the Euclidean distance, a common metric for vector spaces. The implementation is as follows:

```python
def euclidean_distance(vector1, vector2):
    """
    Calculate the Euclidean distance between two vectors.

    Parameters:
    - vector1: First vector (list, numpy array, or similar).
    - vector2: Second vector (list, numpy array, or similar).

    Returns:
    - Euclidean distance between the two vectors.
    """
    vector1 = np.array(vector1)
    vector2 = np.array(vector2)

    # Ensure vectors are of the same length
    if len(vector1) != len(vector2):
        raise ValueError("Vectors must have the same length for Euclidean distance calculation.")

    # Calculate Euclidean distance
    distance = np.linalg.norm(vector1 - vector2)

    return distance

# Example usage
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
euclidean_distance_value = euclidean_distance(vector1, vector2)
```

Listing 9: Euclidean Distance Implementation

The Euclidean distance provides a measure of similarity, with identical vectors resulting in a distance of 0. This implementation is crucial for assessing the dissimilarity between latent vectors during the retrieval process.

4.3 3D Shape Retrieval Implementation

Now equipped with our distance metric, we can proceed with the 3D shape retrieval process. The workflow involves generating 2D views from a new 3D model, extracting the latent vector, and finding the nearest neighbors in our existing database.

```
# Iterate through the test_loader
for sample in test_loader:
    model_3d = sample['path_3d_data_file']
    path_images = sample['path_2d_data_folder']
    labels = sample['label'].to(device)

    result = []
    # Loop over each image in the batch
    for folder_path, label in zip(path_images, labels):
        # Get the list of files in the folder
        image_files = os.listdir(folder_path)

        if image_files:  # Check if there are any files in the folder
            min_distance = float('inf')
            # Load the first image in the folder
            first_image_path = os.path.join(folder_path, image_files[0])
            first_image = Image.open(first_image_path).convert('RGB')

            # Extract latent vector
            latent_vector = features_extractor(transform(first_image).unsqueeze(0).
    to(device)).cpu().detach().numpy().flatten().tolist()

            # Find the closest image in the dataset
            result_info = find_closest_image_info(dataset_latent_vectors,
    latent_vector, first_image_path)
            result.append(list(result_info))
            print(result_info)
    break
```

Listing 10: 3D Shape Retrieval Implementation

The '$find_closest_image_info$' function is responsible for finding the nearest neighbors based on the Euclidean distance metric. Let's include this function:

```
def find_closest_image_info(
    dataset_latent_vectors: Dict[str, Dict[str, List[float]]],
    latent_vector: List[float],
    first_image_path: str
) -> Tuple[float, str, str, str]:
    """
    Finds the closest image information in the dataset_latent_vectors for a given
    latent vector.

    Parameters:
    - dataset_latent_vectors (Dict[str, Dict[str, List[float]]]): Latent vectors
    for images in the dataset.
    - latent_vector (List[float]): Latent vector for the query image.
    - first_image_path (str): Path of the query image.

    Returns:
    - Tuple[float, str, str, str]: Information for the closest image.
      Contains (distance, json_file, image, first_image_path).
      Image is the one in our database
    """
    min_distance = float('inf')
    result_info = []

    for json_file, model_3d in dataset_latent_vectors.items():
        for image, liste in model_3d.items():
            distance = euclidean_distance(liste, latent_vector)

            # Check if the current distance is smaller than the minimum
            if distance < min_distance:
                min_distance = distance
                result_info = [distance, json_file, image, first_image_path]

    return tuple(result_info)
```

Listing 11: Find Closest Image Information Function

The retrieval process allows us to identify the closest matches in our database for a given new 3D model, providing valuable information for shape analysis and recognition.

4.4 Results

To visually assess the effectiveness of our 3D shape retrieval system, we provide two figures. The first figure displays a set of 2D views generated from our test 3D models. The second figure presents the corresponding nearest neighbors from our database based on the extracted latent vectors.

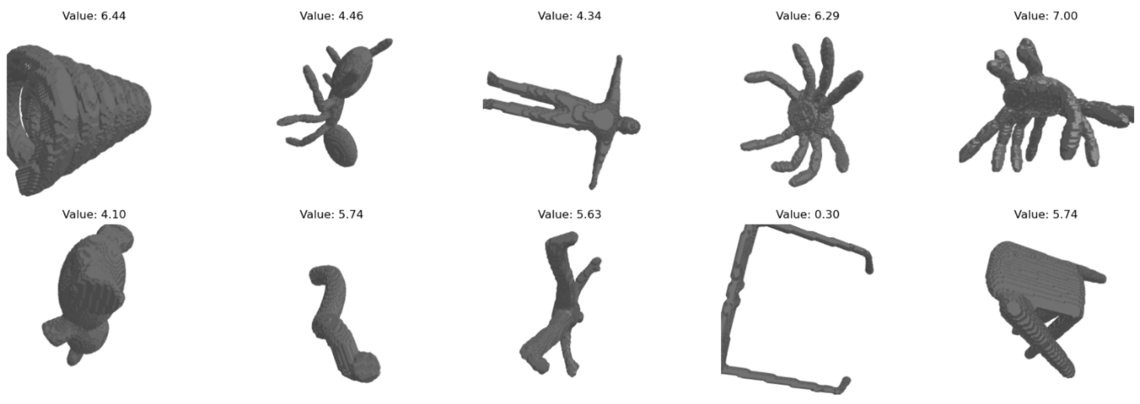


Figure 5: 2D views generated from test 3D models.

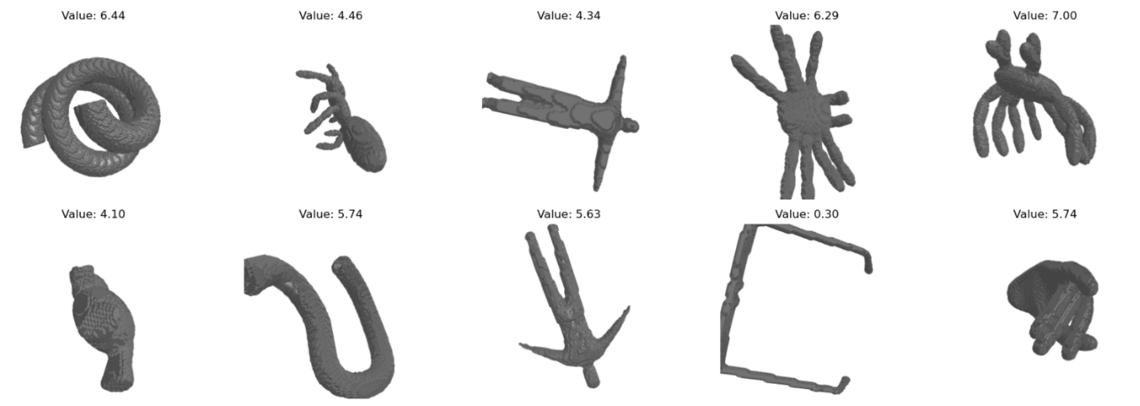


Figure 6: Nearest neighbors from the database corresponding to the test 3D models.

Figure 5 showcases the 2D views generated from our test 3D models, providing an insight into the diversity of shapes in our testing dataset. In Figure 6, we present the 2D views associated with the 3D models from our database, which are identified as the nearest neighbors based on the extracted latent vectors. This comparison allows for a visual evaluation of the success of our 3D shape retrieval system.

4.5 Challenges in Quantifying Retrieval Efficacy

Measuring the effectiveness of our 3D shape retrieval system using quantitative metrics poses challenges due to the inherent ambiguity in determining the ”correct” match for a given 3D model. Unlike typical classification tasks, where a ground truth label is known, the matching process in 3D shape retrieval involves finding similar shapes without predefined class labels.

Two approaches can be considered for assessing retrieval efficacy. The first approach involves a brute-force comparison of the latent vector of the test 3D model with all latent vectors in the database, selecting the model with the closest latent vector. However, this method may be computationally expensive, especially as the database size increases.

A more targeted approach involves classifying the test 3D model using the 2D views generated from it. Subsequently, the retrieval process focuses on finding the nearest neighbors within the same predicted class. While this approach may reduce computational complexity, it introduces potential errors if the initial classification is inaccurate.

Moreover, enhancing the model's performance could involve adopting a multi-view strategy. Instead of relying on a single generated image and its associated latent vector, multiple views of the test 3D model can be considered. Each view contributes its latent vector, and a voting mechanism can be employed to determine the most probable match among the retrieved models.

These considerations highlight the nuanced nature of 3D shape retrieval evaluation, emphasizing the need for a combination of quantitative and qualitative assessments to comprehensively evaluate the system's performance.

5 Conclusion and Additional Resources

In conclusion, this report outlines the development of a 3D shape retrieval system using latent vectors extracted from 2D views of 3D models. The workflow involves generating latent vectors for training models, creating a ResNet-based classifier for feature extraction, and implementing a retrieval application for new 3D models.

For a more in-depth understanding of the codebase and practical usage instructions, a detailed README document is provided alongside this report. The README covers essential information, including project structure, dependencies, code execution, and guidance on utilizing the developed 3D shape retrieval application.

Access the README document for comprehensive insights and practical instructions on the project's codebase.