3D Object Reconstruction Generative Adversarial Network

Repositorio original: https://github.com/renato145/3D-ORGAN

Integrantes

- Anto Chávez, Carolain
- Atarama León, Diego Sebastián
- Yanayaco Temoche, David Martin

Índice

Introducción

Muchas veces un objeto puede dañarse perdiendo así su forma original ocasionando que este sea muy complicado de reconocer. Dicho problema puede aumentar en el ámbito de la excavación donde muchas veces se encuentran objetos dañados difíciles de identificar. Nuestro objetivo es construir un modelo el cual, mediante un análisis geométrico, facilite la reparación de objetos dañados desde una perspectiva computacional, con la finalidad de que pueda ser utilizado en distintos ámbitos, reduciendo así la carga de trabajo y el tiempo de procesamiento.

Archivos relevantes

Repositorio original

Carpeta principal

Los archivos que inicial con reading_3d_files_ inicialmente leen los modelos 3D en diferentes formatos almacenados en el directorio indicado por el usuario; lo convierten al formato BinVox, con el programa instalado[1] en el computador del usuario para ello; y luego lo transforman a un array booleano 3D.

- **reading_3d_files_3DPotteryDataset.ipynb:** Utiliza modelos 3D en formato OBJ; genera un diccionario para la data y muestra los *labels* obtenidos ('Abstract', 'Alabastron', 'Amphora', 'Aryballos', 'Bowl', 'Dinos', 'Hydria', 'Kalathos', 'Kantharos', 'Krater', 'Kyathos', 'Kylix', 'Lagynos', 'Lebes', 'Lekythos', 'Lydion', 'Mastos', 'Modern-Bottle', 'Modern-Glass', 'Modern-Mug', 'Modern-Vase', 'Mug', 'Native American - Bottle', 'Native American - Bowl', 'Native American - Effigy', 'Native American - Jar', 'Nestoris', 'Oinochoe', 'Other', 'Pelike', 'Picher Shaped', 'Pithoeidi', 'Pithos', 'Psykter', 'Pyxis', 'Skyphos').
- **reading_3d_files_Larco.ipynb:** Lee modelos 3D en formato OFF; coloca la data en un diccionario y lo guarda en formato npy (numpy).
- **reading_3d_files_ModelNet.ipynb:** Obtiene data de entrenamiento en formato OFF y la data guardada en formato npy, separa la data de entrenamiento y de validación y la aumenta.
- **run_experiments.sh:** De este archivo solo se destaca que para el modelo usa 400 épocas (*epochs*).
- **simulate_fractures.ipynb:** Utiliza la función get_fractured de reconstruction.utils.data_prep para generar fracturas en la data de validación.
- **view_results.ipynb:** Utiliza tensorflow para backend. Usa la función LoadModel de reconstruction.model para leer el modelo generado; predice doblemente de los voxels generados (para reconstruir las partes que no hubieran sido reconstruidas); compara la pérdida de información por categoría y los voxel adicionales generados.

Carpeta "reconstruction/model"

La carpeta model, dentro de reconstruction cuenta con 5 archivos python (.py). Dentro de estos archivos se encuentran las clases y las funciones necesarias para la creación de un modelo que funcione con el dataset de los objetos 3D. Es decir, el model como tal es creado dentro de esta carpeta. El archivo "__init__.py" es el principal y el que importa a los demás archivos. En este archivo el modelo es creado para ser importado en la solución.

Carpeta "reconstruction/utils"

Dentro de la carpeta utils encontramos que existe una subcarpeta pyntcloud y 5 archivos (.py). El primer archivo que encontramos es el "__init__.py" en el que hay un código para definir el formato de tiempo

[1] Programa para convertir de algunos formatos 3D a Binvox: https://www.patrickmin.com/binvox/

que se utilizara. Luego tenemos el archivo “binvox_rw.py” el cual nos ayudará a relacionar los voxeles de un modelo 32x32x32 a las coordenadas del modelo original. A continuación, tenemos el archivo “data_prep.py” en el que se prepara toda la data para determinar si es esfera o cubo y tiene como datos de entrada Nx3points y como datos de salida unos puntos dentro del rango [-radio, radio]. Para finalizar tenemos el archivo más importante que en este caso es “plot.py” en donde se encuentra la función para dibujar en 3d.

Repositorio propio

Carpeta principal

- **data_display.ipynb:** Lee la data extraída del dataset original y obtiene únicamente la data del tipo de modelos a entrenar (*chair*), luego muestra una visualización de la misma.
- **fracturing_data.ipynb:** Con ayuda del repositorio proporcionado, hemos creado un dataset con más de 7000 sillas, completas y rotas. Es importante aclarar que de sillas completas distintas, solo hay 889, estas se repiten 8 veces cada una en nuestro dataset. Sin embargo, cada silla sí cuenta con 8 formas distintas de fractura. Es decir, sí existen más de 7000 sillas fracturadas de manera distinta. Esto se hizo así para que nuestro modelo tenga mejor precisión.
- **Data.md:** Este archivo .md contiene el enlace a el dataset creado por nosotros.
- **FullAndFracturedVisualization.ipynb:** En este notebook se puede observar una visualización de un ejemplo de nuestra data creada. Se observa una silla y dos fracturas distintas de la misma. Cabe aclarar que para observar las sillas se necesita abrir el archivo en Colab o Jupyter. No se podrán visualizar desde GitHub.

Carpeta data

Contienen carpetas comprimidas de archivos numpy (.npy) con determinada data.

- **Full.zip:** Posee la data del repositorio original filtrada por el tipo elegido por el equipo (en este caso se eligió sillas).
- **FullAndFracture.zip**: Contiene la data duplicada y fracturada.

Algoritmos y modelos

Acorde al archivo environment.yaml, las librerías usadas son:
Keras-Applications (v. 1.0.8), Keras (v. 2.3.1), PyYAML (v. 6.0), click (v. 7.1.2), future (v. 0.18.2), google-pasta (v. 0.2.0), h5py (v. 2.10.0), joblib (v. 1.1.0), libmesh (v. 0.0.0), numpy (v. 1.16.0), protobuf (v. 3.19.1), scikit-learn (v. 0.23.2), scipy (v. 1.5.2), tensorboard (v. 1.14.0), tensorflow-estimator (v. 1.14.0), tensorflow-gpu (v. 1.14.0), termcolor (v. 1.1.0), threadpoolctl (v. 3.0.0) y torch (v. 1.5.0).

Preparación de data

Binvox a Arrays booleanos 3D

Inicialmente se requiere transformar la data en cualquier formato al formato Binvox y posteriormente convertirlo a un array 3D, como se realizó en archivos anteriores.

```
import os
from reconstruction.utils import binvox_rw
from reconstruction.utils.plot import plot_vol

def voxels_from_file(file, voxsize=32):
    out_file = file.split('.')[0] + '.binvox'
    file = file.replace(' ', '\ ')
    cmd = f'tools/binvox/binvox -d {voxsize} -cb -e {file}'

    if os.path.exists(out_file):
        os.remove(out_file)

    t = os.system(cmd)

    if t == 0:
        with open(out_file, 'rb') as f:
            d = binvox_rw.read_as_3d_array(f).data
            d = np.transpose(d, (0, 2, 1)) # fix orientation

        os.remove(out_file)
        return 1, d, file.split('/')[-1]
    else:
        return 0, None, None
```

La función **voxels_from_file** del repositorio original, ya que ejecuta la acción anteriormente mencionada. Esta recibe como parámetros el nombre del archivo y el tamaño del voxel requerido (es decir la cantidad de capas, filas y columnas que tendrá el array 3D). Luego lee el formato del archivo; genera el nuevo nombre (que será igual al original salvo la extensión del mismo); lo convierte a binvox; si se ejecutó la operación, lo convierte a un array 3D de booleanos y devuelve 1 (se ejecutó correctamente), el array 3D y el nombre del archivo.

Luego, usaremos la función para obtener el array mencionado anteriormente, utilizaremos funciones del archivo binvox_rw.py:.

```
import numpy as np

class Voxels(object):
    def __init__(self, data, dims, translate, scale, axis_order):
        self.data = data
        self.dims = dims
        self.translate = translate
        self.scale = scale
        assert (axis_order in ('xzy', 'xyz'))
        self.axis_order = axis_order

    def clone(self):
        data = self.data.copy()
        dims = self.dims[:]
        translate = self.translate[:]
        return Voxels(data, dims, translate, self.scale, self.axis_order)

    def write(self, fp):
        write(self, fp)
```

```
def read_header(fp):
    line = fp.readline().strip()
    if not line.startswith(b'#binvox'):
        raise IOError('Not a binvox file')
    dims = list(map(int, fp.readline().strip().split(b' ')[1:]))
    translate = list(map(float, fp.readline().strip().split(b' ')[1:]))
    scale = list(map(float, fp.readline().strip().split(b' ')[1:]))[0]
    line = fp.readline()
    return dims, translate, scale


def read_as_3d_array(fp, fix_coords=True):
    dims, translate, scale = read_header(fp)
    raw_data = np.frombuffer(fp.read(), dtype=np.uint8)
    values, counts = raw_data[::2], raw_data[1::2]
    data = np.repeat(values, counts).astype(np.bool)
    data = data.reshape(dims)
    if fix_coords:
        # xzy to xyz
        data = np.transpose(data, (0, 2, 1))
        axis_order = 'xyz'
    else:
        axis_order = 'xzy'
    return Voxels(data, dims, translate, scale, axis_order)
```

La clase **Voxels** organiza la data enviada y confirma si los ejes están colocados correctamente. Mientras que la función **read_header** recibe un nombre de archivo, valida que indique que es un archivo Binvox, y retorna las dimensiones, traslación y escala. Finalmente, la función **binvox_rw.read_as_3d_array** recibe las dimensiones, traslación y escala de la función read_header y la data cruda de la lectura del archivo con numpy. Esta data indica los valores y la cantidad de veces que se repite, por lo que es transformada a un array booleano considerando dichos valores y repeticiones; luego remodela el array en las dimensiones indicadas (32, 32, 32); y por último, de ser indicado, transforma de las coordenadas xzy a xyz.

Fracturación de los objetos

Dado que el proyecto consiste en facilitar la reconstrucción de objetos, además de un dataset de objetos completos, se requería que estos tuvieran fracturas. De esta manera, los objetos fracturados serían utilizados como dominio de la función target (X) y los objetos completos, como target (Y). Es por ello que para obtener la data a enviar al modelo, se generaron fracturas en todos los objetos.

```
def get_fractured(source, min_points=1, max_points=4, min_radius=3,
max_radius=6, sphere_chance=0.75):
    fractured = source.copy()
    points = np.random.randint(min_points, max_points+1)
    idxs = np.argwhere(fractured == 1)
    centers = idxs[np.random.choice(len(idxs), points, False)]

    for x,y,z in centers:
        r = np.random.randint(min_radius, max_radius+1)
        xmin, xmax = max(0, x-r), x+r
        ymin, ymax = max(0, y-r), y+r
        zmin, zmax = max(0, z-r), z+r
```

```
        sphere = np.ones_like(fractured)
        sphere[xmin:xmax, ymin:ymax, zmin:zmax] = 0
        # sphere or cube
        if np.random.rand() < sphere_chance:
            idxs = np.argwhere(sphere == 0)
            idxs_remove = np.sqrt((idxs[:,0] - x)**2 + (idxs[:,1] -
y)**2 + (idxs[:,2] - z)**2)
            idxs_remove = idxs[idxs_remove > r]
            sphere[idxs_remove[:,0], idxs_remove[:,1],
idxs_remove[:,2]] = 1

        fractured *= sphere

    return fractured
```

La función **get_fractured** recibe como parámetro principal un array booleano 3D; genera una copia del mismo, una determinada cantidad de puntos aleatorios (por defecto entre 1 a 4); lee y guarda como índices las partes donde el array es verdadero (es decir, es parte del objeto); y obtiene puntos centrales aleatorios de fractura utilizando como base los índices guardados y la cantidad de puntos. Luego por cada punto central aleatorio establece un radio (por defecto entre 3 a 6), y genera una cubo de puntos que convierte a 0, fracturando el objeto; además, aleatoriamente decide si aumentará la fractura a una forma esférica, si ello ocurre, algunos puntos adicionales serán convertidos a 0.

Ploteo y Visualización

El hecho de contar con los datos en formato Binvox nos facilita el ploteo de los objetos. Aún así, para tener un ploteo más completo (con movimiento del pov), es necesario importar unas librerías extras para ello, como matplotlib.pyplot, plotly.graph_objs y iplot (desde plotly.offline).

```
def plot3d(verts, s=10, c=(105,127,155), show_grid=False):
    x, y, z = zip(*verts)
    color = f'rgb({c[0]}, {c[1]}, {c[2]})'
    trace = go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        marker=dict(
            size=s,
            color=color,
            line=dict(
                color='rgba(217, 217, 217, 0.14)',
                width=0.5
            ),
            opacity=1
        )
    )
    data = [trace]
    layout = go.Layout(
        margin=dict(l=0, r=0, b=0, t=0),
        scene = go.Scene(
            xaxis=dict(visible=show_grid),
            yaxis=dict(visible=show_grid),
            zaxis=dict(visible=show_grid)
```

```
        )
    )
    fig = go.Figure(data=data, layout=layout)
    iplot(fig)
```

```
def point_cloud_to_volume(points, vsize, radius=1.0):
    vol = np.zeros((vsize,vsize,vsize), dtype=np.bool)
    voxel = 2*radius/float(vsize)
    locations = (points + radius)/voxel
    locations = locations.astype(int)
    vol[locations[:,0],locations[:,1],locations[:,2]] = 1.0

    return vol
```

```
def volume_to_point_cloud(vol):
    """ vol is occupancy grid (value = 0 or 1) of size
vsize*vsize*vsize
        return Nx3 numpy array.
    """
    vsize = vol.shape[0]
    assert(vol.shape[1] == vsize and vol.shape[1] == vsize)
    points = []
    for a in range(vsize):
        for b in range(vsize):
            for c in range(vsize):
                if vol[a,b,c] == 1:
                    points.append(np.array([a,b,c]))
    if len(points) == 0:
        return np.zeros((0,3))
    points = np.vstack(points)

    return points
```

```
def auto_pcl_to_volume(points, vsize):
     data_min = np.min(points)
     data_max = np.max(points)
     radius = max(abs(data_min), data_max)
     radius = math.ceil(radius*100) / 100
     vol = point_cloud_to_volume(points, vsize, radius)

     return vol
```

```
def plot_vol(vol, s=10, c=(105,127,155), show_grid=False):
    if vol.dtype != np.bool:
        vol = vol > 0

    pc = volume_to_point_cloud(vol)
```

```
    plot3d(pc, s, c, show_grid)
```

Estas 5 funciones, del repositorio proporcionado, son utilizadas en nuestro proyecto para mostrar las sillas, permitiendo girar el objeto para observarlo desde diferentes puntos de vista.

Data utilizada

Utilizaremos inicialmente la data obtenida del repositorio (que se halla dentro de un archivo tar, en formato numpy) https://github.com/renato145/3D-ORGAN/blob/master/datasets/arq_dataset.tar.gz?raw=true. Posteriormente, adicionaremos data a este dataset utilizando modelos 3D de https://www.polantis.com/ikea.

Configuración de la data

La data obtenida del archivo numpy se mantendrá como array booleano 3D; mientras que la data nueva será transformada con el programa **[binvox] 3D mesh voxelizer**[2], el cual primero será descargado en el computador, al formato Binvox y posteriormente se transformará a un array booleano 3D.

Primero, separaremos la data que nos interesa. En nuestro caso, las sillas, estas se encuentran como clase 'chair'. Luego, debido a que solo contamos con 889 sillas, multiplicaremos esta cantidad por 8. Así obtendremos más datos para que nuestro modelo entrene. Después, con ayuda de la función get_fractured, obtendremos sillas rotas a partir de las originales. Esto nos dará un dataset con 889 sillas distintas y 8 versiones de esas sillas rotas. Con este dataset final, que cuenta con más de 7000 sillas (889 únicas, 8 versiones diferentes de silla rota por cada una), nuestro modelo podrá entrenar para reconstruirlas.

[2] https://www.patrickmin.com/binvox/